Д.В. Иртегов

Многопоточное программирование с использованием POSIX Threads



ИНТУИТ
НАЦИОНАЛЬНЫЙ ОТКРЫТЫЙ УНИВЕРСИТЕТ

Многопоточное программирование с использованием POSIX Threads

2-е издание, исправленное

Иртегов Д.В.

Национальный Открытый Университет "ИНТУИТ"
2016

Многопоточное программирование с использованием POSIX Threads/ Д.В. Иртегов - М.: Национальный Открытый Университет "ИНТУИТ", 2016

Курс посвящен разработке многопоточных приложений на языках C/C++ с использованием POSIX Thread Library.

Основное внимание уделяется разработке ориентированных на ввод-вывод приложений и сетевых сервисов, но рассматриваются также некоторые вопросы, связанные с разработкой вычислительных программ с разделяемой памятью и приложений реального времени. Примеры программ тестировались под ОС Solaris 10 с использованием Sun Studio 11.

(c) ООО "ИНТУИТ.РУ", 2010-2016
(c) Иртегов Д.В., 2010-2016

Введение

Задача этого курса – обучить вас разработке многопоточных приложений для Sun Solaris 10. Значительная часть полученных знаний может быть использована для разработки многопоточных приложений для других систем семейства Unix, поддерживающих POSIX threads API. Это такие системы, как Linux (начиная с версии 2.4), Free BSD, SCO Unixware, IBM AIX и др.

Практические задания к курсу скачать: http://old.intuit.ru/department/se/posixthreads/1/Practice.zip

1. Что такое потоки

Теперь у нее есть дочь. Другое поколение,другие дела

Ей только пять лет, но время летит как стрела

И хотя она пока что не умеет читать

Она уже знает больше, чем знала мать

Ведь она видит сразу много программ

Глядя в телевизор

Б. Гребенщиков

В Unix-системах для исполнения программ используются объекты, называемые процессами. Процессу выделяется собственное виртуальное адресное пространство и некоторые другие ресурсы. Для каждого процесса создается иллюзия последовательного исполнения. Программа исполняется в рамках процесса так же, как она исполнялась бы в однозадачной ОС.

Процесс взаимодействует с ядром ОС при помощи так называемых системных вызовов. При исполнении системного вызова, процесс исполняет специальную команду (у современных версий x86 эта команда называется SYSCALL, у SPARC – TA 0x8 для 32-битных

программ, TA 0x40 для 64-битных программ), которая переключает адресное пространство и передает управление ядру. Процессы в традиционных Unix-системах могут взаимодействовать друг с другом только при помощи системных вызовов – чтения и записи в трубы и разделяемые файлы, а в более современных системах – при помощи System V IPC.

Таким образом, процессы в Unix надежно изолированы друг от друга. Нарушения целостности данных одного процесса (например, в результате переполнения буфера или ошибок при работе с указателями) приводят к аварийному завершению этого процесса, но не затрагивают, во всяком случае, непосредственно, другие процессы. Даже удаленное исполнение кода в рамках одного из процессов приводит только к тому, что злоумышленник, написавший этот код, получает лишь те привилегии, которые имел этот процесс.

Однако ряд задач – примеры которых мы рассмотрим далее в этом разделе – требует реализации в виде нескольких параллельно (или, на однопроцессорной машине, квазипараллельно) исполняющихся процессов. Традиционные Unix-системы предполагали решать такие задачи при помощи нескольких взаимодействующих процессов. Среди системных вызовов, поддерживаемых всеми современными Unix-системами можно выделить несколько групп, специально предназначенных для такого взаимодействия – это вызовы для работы с трубами (pipe), сетевые средства (которые можно применять и для взаимодействия между процессами на одной машине) и System V IPC. К сожалению, далеко не для всех задач эти средства оптимальны, а для некоторых и вовсе неадекватны.

Кроме того, нередко возникала потребность в переносе приложений, разработанных для ОС, допускавших несколько нитей исполнения в пределах одной задачи или процесса, под Unix. Поэтому в современных Unix-системах было введено понятие нитей (thread), которые соответствуют единицам планирования в рамках одного процесса. Нити разделяют общее адресное пространство, но планируются независимо. Иллюзия последовательного исполнения создается для нити, а не для процесса в целом.

В начале 90х в различных Unix-системах использовались разные

подходы к реализации многопоточности и несовместимые API для управления потоками. В 1995 году был принят стандарт IEEE POSIX 1003.1c-1995 (также известен как ISO/IEC 9945-1:1996). Большинство современных Unix-систем, а также некоторые не-Unix системы (например, OpenVMS и IBM zOS) реализуют этот стандарт.

При разработке новых приложений рекомендуется пользоваться стандартным API. Это облегчит вам перенос приложений под другие ОС и под новые версии Solaris.

2. Зачем нужны многопоточные программы

1. Улучшение времени реакции интерактивных программ.
2. Улучшение времени реакции серверных приложений. Возможность обрабатывать несколько запросов одновременно.
3. Использование дополнительных ресурсов на многопроцессорных и гипертрединговых компьютерах.
4. Задачи реального времени

2.1 Улучшение времени реакции интерактивных программ

1. Фоновое скачивание страницы в браузере
2. Фоновый ввод-вывод (например, утилита просмотра файла может считывать файл по мере его просмотра)
3. Фоновая проверка орфографии
4. Фоновое переразбиение текста на страницы в WYSIWYG текстовых процессорах.

В старых версиях MS Word, переразбиение текста на страницы осуществлялось по мере набора текста, так что вставка текста в начало документа из нескольких страниц превращалась в чистое издевательство над пользователем.

Незнакомые с техникой подготовки печатных документов читатели, возможно, нуждаются в пояснении, что такого сложного в задаче разбиения текста на страницы. Дело в том, что в типографской практике

хорошим тоном считается минимизировать количество разбиений параграфов при переносе со страницы на страницу. Также, ряд явлений, возникающих при наивном разбиении текста на страницы, например "висячие строки", когда на страницу попадает только одна строка из параграфа, или "висячие заголовки", когда заголовок раздела попадает на конец страницы, а тело раздела размещается на следующей, считаются совершенно нетерпимыми.

При этом также считается хорошим тоном делать текст на страницах по возможности одинаковой высоты. В условиях, когда текст содержит крупные неразбиваемые объек-ты, такие как рисунки и таблицы, это требование может оказаться трудновыполнимым. Во всяком случае, минимизация "дефекта" заполнения страниц сводится к так называемой задаче о рюкзаке, которая относится к классу NP-полных задач (т. е. задач, для которых неизвестно решения, работающего быстрее, чем полный перебор вариантов). На практике, системы подготовки печатных документов — как TeX и troff, так и WYSIWYG-системы — ищут лишь относительно приемлемое решение, а для подготовки документа типографского качества, т. е. такого, который не стыдно издать в виде книги, нуждаются в помощи верстальщика. Тем не менее, даже поиск относительно приемлемого решения в сложных случаях может продолжаться десятки секунд.

В Microsoft Word для Win 32, задача переразбиения текста на страницы решается фоновой нитью, которая работает параллельно с пользовательским интерфейсом. При нормальной работе пользователь этого вообще не замечает (впрочем, внимательный пользователь может увидеть, что при редактировании в нижней панели возникает анимационная иконка в виде перелистываемой книги, на которой что-то пишет карандаш). Фактически, именно введение фонового разбиения на страницы в версии Office 95 сделало Word более или менее пригодным для редактирования больших документов.

В StarOffice / OpenOffice переразбиение текста на страницы также осуществляется в фоновом режиме при помощи отдельной нити.

2.2 Улучшение времени реакции серверных приложений

Главные характеристики большинства серверов приложений – это среднее время исполнения запроса и количество запросов, обрабатываемых в единицу времени. Это две разные характеристики.

Среднее время исполнения запроса – это основная характеристика производительности сервера с точки зрения отдельного пользователя. Увеличение этого времени ведет к снижению производительности пользователя, ему приходится тратить больше времени на исполнение тех же самых задач. Кроме того, слишком большое время реакции системы попросту раздражает пользователей.

Количество запросов, обрабатываемых в единицу времени – это основная характеристика производительности сервера с точки зрения владельца этого сервера. Если мы найдем способ увеличить это количество, не ухудшив или незначительно ухудшив показатели исполнения отдельных запросов, мы сможем подключить к серверу больше пользователей. Таким образом, владелец сервера может сэкономить на приобретении оборудования.

Рассмотрим процесс исполнения одиночного запроса типичным серверным приложением (см. рис. 1.1). Он состоит из пяти этапов: приема запроса (выполняется сетевым интерфейсом), анализа запроса (выполняется центральным процессором), считывания данных с диска (выполняется дисковым контроллером), создания ответа (выполняется центральным процессором) и передачи ответа. Разумеется, времена исполнения этих этапов могут различаться в зависимости от типа сервиса, предоставляемого сервером, и от самих запросов. Для сервера http, раздающего статические HTML страницы, время анализа запроса и форматирования ответа будет очень малым по сравнению с временами чтения с диска и передачи данных по сети. Напротив, у сервера приложений основная нагрузка ляжет на центральный процессор. Серверу баз данных может потребоваться несколько обращений к диску для того, чтобы найти все относящиеся к запросу данные, и т.д.. Тем не менее, давайте рассмотрим условный сервер, обрабатывающий однотипные запросы, у которых структура обработки запроса именно такова, как на рис. 1.1, а времена исполнения всех этапов одинаковы (последнее требование нужно лишь для того, чтобы мне было проще рисовать картинку).



Рис. 1.1. Исполнение одиночного запроса серверным приложением

Рассмотрим теперь исполнение потока запросов. Однопоточный сервер должен был бы исполнять запросы строго последовательно, поэтому максимальное количество запросов, исполняемых в секунду, было бы равно `1/t`, где `t` – время исполнения одиночного запроса.

При этом среднее время исполнения запроса не будет равно `t`, а будет расти в зависимости от вероятности перекрытия запросов во времени. Теоретико-вероятностные расчеты и практика показывают, что когда поток запросов приближается к `1/t` в секунду, эта вероятность становится весьма значительной, так что время исполнения одиночного запроса может увеличиться во много раз. Однако к многопоточному серверу эти расчеты неприменимы. Рассмотрим, как многопоточный сервер мог бы обрабатывать поток запросов (см. рис. 1.2).

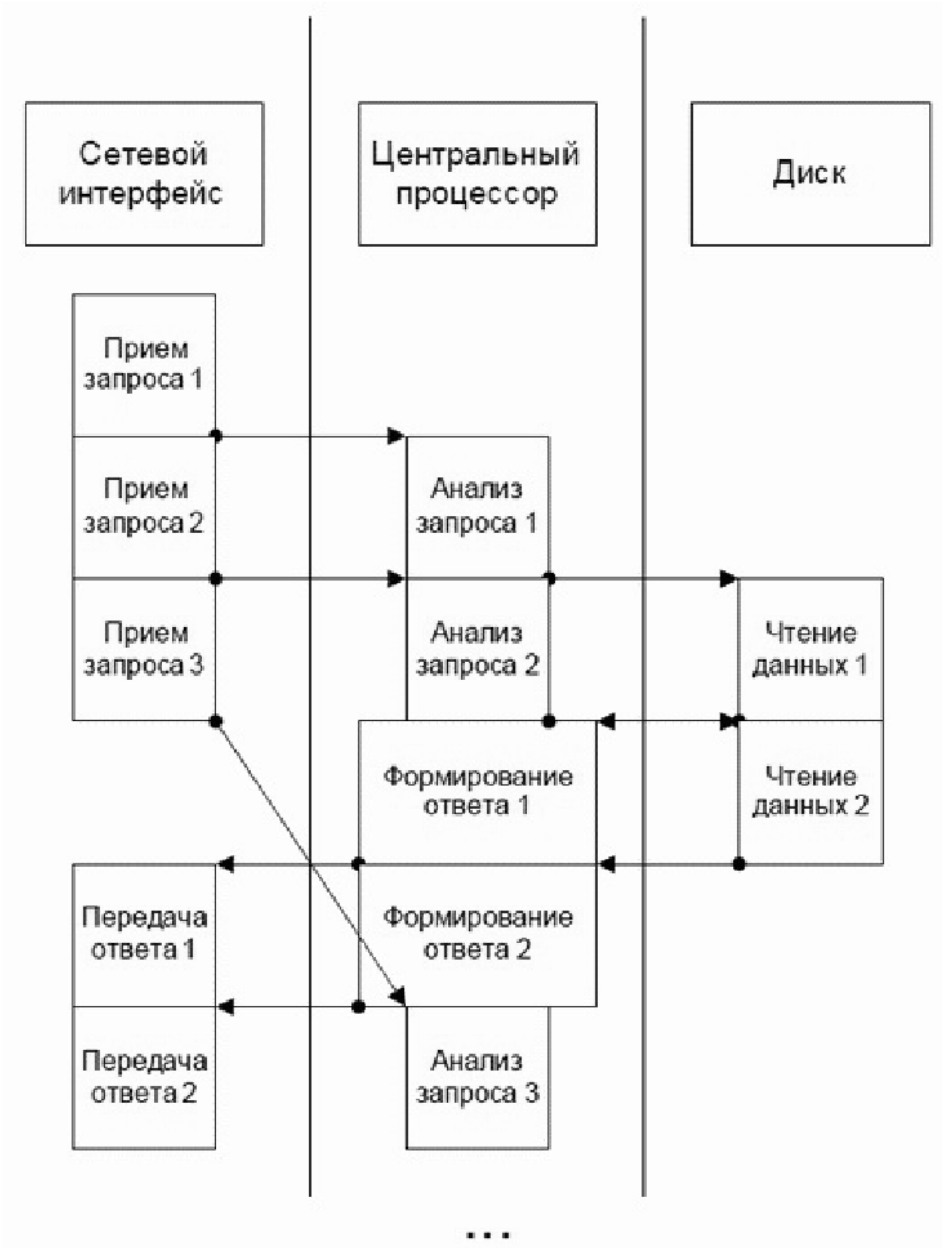


Рис. 1.2. Исполнение потока запросов многопоточным сервером

Видно, что сервер совмещает этапы исполнения перекрывающихся во времени запросов. Если относительные времена исполнения этапов запросов таковы, как на рис. 1.1 и рис. 1.2, избежать взаимодействия

запросов не удается, так что среднее время исполнения запроса может вырасти по сравнению с `t`. Но количество запросов, исполняемых в секунду, оказывается значительно больше, чем `1/t`.

Более того, видно, что мы могли бы повысить производительность системы, установив второй процессор и второй сетевой интерфейс.

Однако переделка однопоточного серверного приложения в многопоточное требует специальной поддержки со стороны операционной системы и значительной переработки кода приложения. Все современные ОС общего назначения – Sun Solaris, другие системы семейства Unix, IBM zOS, Win 32 и Win 64 – обеспечивают соответствующую поддержку, но далеко не все разработчики приложений умеют этой поддержкой пользоваться.

3. Проблемы многопоточности

Первые попытки организации параллельных вычислений предпринимались еще в 60е годы XX столетия. Теория межпроцессного и многопоточного взаимодействия была в основном разработана еще тогда, однако массовое распространение многопоточное программирование получило лишь в 90е годы. Это было обусловлено рядом причин, основные из которых перечислены ниже:

1. Несовместимость со старыми (однопоточными) компиляторами
2. Несовместимость со старыми библиотеками
3. Несовместимость или ограниченная поддержка многопоточных программ другими инструментальными средствами, в первую очередь отладчиками
4. Несовместимость многих принятых практик программирования с многопоточностью.

Первые три из названных причин чисто технические, для их решения требуется переработка инструментальных средств разработки программ. Однако, поскольку такая переработка нарушает совместимость, массовый переход на новый инструментарий занял многие годы. К тому же, поскольку первые версии многопоточных инструментальных средств обладали различными недостатками, это также затрудняло их

принятие разработчиками.

Первые среды программирования, реализовавшие многопоточность, такие, как Simula 67, использовали для создания потоков и межпоточного взаимодействия специальные конструкции языка. Эта традиция продолжалась и до 80х годов и воплощена в таких языках, как Ada, Occam, Parallel Fortran. Пожалуй, только последний из этих языков – Parallel Fortran – получил широкое практическое применение.

При программировании на C / C++ с использованием POSIX Thread Library специальные компиляторы не требуются. POSIX Thread Library можно использовать с любыми компиляторами, реализующими соглашения о вызовах, соответствующие ABI вашей аппаратной платформы. Так, в Solaris 10 многопоточные программы можно писать как на GNU C, так и с помощью Sun Studio C compiler. Все, необходимое для поддержки многопоточности, реализовано на уровне библиотек.

Sun Studio 11 C compiler также поддерживает средства для параллельного программирования OpenMP. При компиляции с ключом `-xopenmp` включается поддержка директив параллелизации OpenMP в исходном коде программы. При компиляции с ключом -xautopar компилятор пытается автоматически найти параллелизуемые участки в программе и реализовать их многопоточное исполнение. OpenMP в Solaris реализован на основе тех же технологий, что и POSIX Threads, поэтому следует проявлять осторожность при совместном использовании OpenMP и POSIX Threads в одной программе. В рамках данного курса OpenMP не рассматривается. Более сложной проблемой на практике оказалась поддержка многопоточности на уровне библиотек. Проблемы, возникающие со старыми библиотеками, в основном описываются на следующей лекции, но полный масштаб этих проблем станет вам понятен лишь после завершения курса в целом. Первые реализации стандартных библиотек C и Fortran для Unix-систем были рассчитаны на однопоточное исполнение, поэтому перенос в многопоточную среду потребовал их переработки. Первые версии C и C++ компиляторов для старых Unix-систем (как и старые реализации C / C++ для OS/2 и Win 32) использовали две разные версии библиотек для однопоточных и многопоточных программ.

В Solaris 10 в основном завершена переработка стандартных библиотек

языка C для оптимальной поддержки многопоточности. Стандартная библиотека `/usr/lib/libc.so` и основная масса других библиотек, входящих в стандартную поставку системы, работают как в однопоточных, так и в многопоточных программах. Тем не менее, эта поддержка до сих пор сопряжена с некоторыми ограничениями. Многие функции, в том числе некоторые функции стандартных библиотек ANSI/ISO C в многопоточных программах следует использовать с осторожностью. Необходимая информация приводится в секции ATTRIBUTES страниц системного руководства (man(1)) по соответствующим функциям. Подробнее этот вопрос обсуждается на следующей лекции.

При использовании библиотек других поставщиков необходимо получить информацию о поддержке многопоточности у поставщика библиотеки. Обычно эта информация должна указываться в документации по библиотеке, но, к сожалению, не все разработчики библиотек это делают. Поддержка многопоточной отладки – также относительно недавнее достижение. В современных версиях популярного отладчика gdb (GNU Debugger) многопоточность поддерживается, но при сборке отладчика ее можно выключить, поэтому до сих пор нередко можно встретить бинарный дистрибутив gdb без поддержки многопоточности. Для проверки, поддерживает ли ваша версия gdb многопоточность, следует исполнить команду info threads. Если поддержка включена, эта команда выводит список нитей отлаживаемой программы, если выключена — команда игнорируется отладчиком.

```
gdb) info threads
    3 process 35 thread 27 0x34e5 in sigpause ()
    2 process 35 thread 23 0x34e5 in sigpause ()
  * 1 process 35 thread 13 main (argc=1, argv=0x7fffff8) at threadtest.c:68
```

Пример 1.1.

В поставку Sun Studio 11 входит отладчик dbx, который полностью поддерживает многопоточную отладку как в режиме командной строки, так и в экранном режиме под управлением Sun Studio IDE. Некоторые сведения об этом отладчике сообщаются в нашем курсе.

Реализации POSIX Threads API

Рассматриваются различные подходы, которые допускает стандарт POSIX, к реализации многопоточности в рамках одного процесса.

Так и живем, не пропустив ни дня,

И каждый день проходит словно дважды.

Б. Гребенщиков

Стандарт POSIX допускает различные подходы к реализации многопоточности в рамках одного процесса. Возможны три основных подхода:

1. В пользовательском адресном пространстве, когда нити в пределах процесса переключаются собственным планировщиком
2. Реализация при помощи системных нитей, когда переключение между нитями осуществляется ядром, так же, как и переключение между процессами.
3. Гибридная реализация, когда процессу выделяют некоторое количество системных нитей, но процесс имеет собственный планировщик в пользовательском адресном пространстве.

Как правило, при этом количество пользовательских нитей в процессе может превосходить количество системных нитей.

1. Пользовательские нити

Реализация планировщика в пользовательском адресном пространстве не представляет больших сложностей; наброски реализаций таких планировщиков приводятся во многих учебниках по операционным системам, в том числе в [Иртегов 2002]. Учебная многозадачная ОС Minix может быть собрана и запущена в виде задачи под обычной Unix-системой; при этом процессы Minix будут с точки зрения ядра системы-хозяина пользовательскими нитями. Главным достоинством пользовательского планировщика считается тот факт, что он может быть реализован без изменений ядра системы.

При практическом применении такого планировщика, однако, возникает серьезная проблема. Если какая-то из нитей процесса исполняет блокирующийся системный вызов, блокируется весь процесс. Устранение этой проблемы требует серьезных изменений в механизме взаимодействия диспетчера системных вызовов с планировщиком операционной системы. То есть главное достоинство пользовательского планировщика при этом будет утеряно. Другим недостатком пользовательских нитей является то, что они не могут воспользоваться несколькими процессорами на многопроцессорной машине – ведь процесс всегда планируется только на одном процессоре!

Наиболее известная реализация пользовательских нитей – это волокна (fibers) в Win32. Считается, что волокна дешевле системных нитей Win32, хотя данных, подтверждающих это утверждение практическими измерениями, у меня нет. Волокна могут использоваться совместно с системными нитями Win32 , но при этом волокна привязаны к определенной нити и исполняются только в контексте этой нити. Волокна не должны исполнять блокирующиеся системные вызовы; попытка сделать это приведет к блокировке нити. Некоторые системные вызовы Win32 имеют неблокирующиеся аналоги, предназначенные для использования в волокнах, но далеко не все. Это резко ограничивает применение волокон в реальных приложениях.

2. Системные нити

Ядро типичной современной ОС уже имеет планировщик, способный переключать процессы. Переделка этого планировщика для того, чтобы он мог переключать несколько нитей в пределах одного процесса, также не представляет больших сложностей. При этом возможны два подхода к такой переделке.

В рамках первого подхода, системные нити выступают как подчиненная по отношению к процессу сущность. Идентификатор нити состоит из идентификатора родительского процесса и собственного идентификатора нити. Идентификатор нити локален по отношению к процессу, т.е. нити разных процессов могут иметь одинаковые идентификаторы. Такой подход реализует большинство систем, реализующих системные нити – IBM MVS-OS/390-zOS, DEC VAX/VMS- OpenVMS, OS/2, Win32, многие Unix-системы, в том числе и

Solaris. В Solaris и других Unix системах (IBM AIX, HP/UX) системные нити называются LWP (Light-Weight Process, "легкие процессы").

Solaris 10 использует системные нити, так что каждой нити POSIX Threads API соответствует собственный LWP. Старые версии Solaris использовали гибридный подход, который рассматривается в следующем разделе.

В рамках другого подхода, системные нити являются сущностями того же уровня, что процесс. Иногда все, что объединяет нити одного процесса – это общее адресное пространство. Наиболее известная ОС, использующая такой подход – Linux. В Linux, нити выглядят как отдельные записи в таблице процессов и отдельные строки в выводе команд top(1) и ps(1), имеют собственный идентификатор процесса.

В старых версиях Linux это приводило к своеобразным проблемам при реализации POSIX Threads API; так, в большинстве Unix-систем завершение процесса системным вызовом exit(2) приводит к немедленному завершению всех его нитей; в Linux вплоть до 2.4 завершалась только текущая нить. В Linux 2.6 был внесен ряд изменений в ядро, приблизивших семантику многопоточности к стандарту POSIX. Эти изменения известны как NPT (Native POSIX Threads).

Наш курс рассчитан на стандартную семантику POSIX Threads API. При программировании для старых (2.4 и младше) версий ядра Linux необходимо изучить особенности поведения этих систем по документации, поставляющейся с системой, или по другим источникам.

3. Гибридная реализация

В гибридной реализации многопоточный процесс имеет несколько LWP и планировщик в пользовательском адресном пространстве. Этот планировщик переключает пользовательские нити между свободными LWP, подобно тому, как системный планировщик в многопроцессорной системе переключает процессы и системные нити между свободными процессами. При этом, как правило, процесс имеет больше пользовательских нитей, чем у него есть LWP.

Причина, по которой этот подход нашел практическое применение – это

убеждение разработчиков первых многопоточных версий Unix, что пользовательские нити дешевле системных, требуют меньше ресурсов для своего исполнения.

При планировании пользовательских нитей возникает проблема блокирующихся системных вызовов. Когда какая-то нить вызывает блокирующийся системный вызов, соответствующий LWP блокируется и на некоторое время выпадает из работы. В старых версиях Solaris эта проблема решалась следующим образом: многопоточная библиотека всегда имела выделенную нить, которая не вызывала блокирующихся системных вызовов никогда. Когда ядро системы обнаруживало, что все LWP процесса заблокированы, оно посылало процессу сигнал SIGWAITING. Библиотечная нить перехватывала этот сигнал и, если это допускалось настройками библиотеки, создавала новый LWP.

Таким образом, если все пользовательские нити исполняли блокирующиеся системные вызовы, то количество LWP могло сравняться с количеством пользовательских нитей. Можно предположить, что компания Sun отказалась от гибридной реализации многопоточности именно потому, что обнаружилось, что такое происходит со многими реальными прикладными программами.

В старых версиях Solaris поддерживался довольно сложный API, позволявший управлять количеством LWP и политикой планирования нитей между ними. Так, можно было привязать нить к определенному LWP. Этот API был частью Solaris Native Threads и нестандартным расширением POSIX Threads API. В рамках данного курса этот API не изучается. Многие современные Unix-системы, в том числе IBM AIX, HP/UX, SCO UnixWare используют гибридную реализацию POSIX Thread API.

4. Сборка приложений с POSIX Threads

Большинство систем, реализующих POSIX Threads, требуют сборки многопоточной программы с библиотекой `libpthread.so` или `libpthread.a`. Как правило, это достигается запуском компилятора с ключом `-lpthread`.

Примечание

Большинство C и C++ компиляторов интерпретируют ключ `-l` следующим образом. К параметру ключа (у ключа `-lpthread` параметром является строка pthread) спереди добавляется строка `lib`, а сзади – строка `.so` или `.a`, в зависимости от того, какой режим сборки задан другими ключами – статический или динамический. Таким образом получается строка `libpthread.so`. Затем файл с таким именем ищется в каталогах, перечисленных в переменной среды LIBPATH. Если эта переменная не установлена, используются каталоги `/lib`, `/usr/lib`, и, возможно, некоторые другие каталоги, зашитые в компилятор.

Так, компилятор Sun Studio 11 при установке по умолчанию ищет дополнительные библиотеки в каталоге `/opt/SUNWspro/lib`.

Компилятор GCC, входящий в поставку Solaris 10, ищет дополнительные библиотеки в каталоге `/usr/sfw/lib` ; в действительности, при сборке GCC ему можно указать пути к дополнительным библиотекам; при сборке GCC из исходных текстов по умолчанию он настраивается на размещение драйвера компилятора (команд gcc и g++) в `/usr/local/bin`, а библиотек – в каталоге `/usr/local/lib`. Эти каталоги можно изменять параметрами скрипта configure, который необходимо запустить перед началом сборки компилятора.

В Solaris 10 ключ `-lpthread` использовать не обязательно – все функции POSIX Thread API включены в стандартную библиотеку языка C `libc.so`, которая подключается по умолчанию. Для совместимости со старыми сборочными скриптами в поставку Solaris 10 также включена пустая библиотека libpthread.so, содержащая ссылки на соответствующие функции в `libc.so`.

Внимание

В некоторых дистрибутивах Linux, библиотека `libstdc++.so` (стандартная библиотека языка C++) содержит ссылки на функции `libpthread.so`. Чтобы облегчить сборку однопоточных программ с такой библиотекой, в библиотеку `libstdc++.so` были включены пустые функции, одноименные функциям POSIX Thread API. При сборке многопоточных программ с такой библиотекой необходимо обязательно

указывать ключ `-lpthread`. Без этого ключа программа соберется (редактор связей не выдаст сообщений о неопределенных символах), но работать не будет (вызов функций POSIX Thread API приведет к ошибке сегментации).

Кроме того, многие компиляторы – в том числе и Sun Studio 11 – рекомендуют компилировать все модули, входящих в многопоточную программу, с ключом `-mt`. В старых системах это могло быть жизненно необходимо, так как в зависимости от наличия или отсутствия этого ключа компилятор мог подключать разные версии стандартной библиотеки времени исполнения. В Solaris 10 этот ключ отвечает только за определение некоторых препроцессорных символов. Впрочем, в следующем разделе мы увидим, что некоторые эти символы также важны при сборке некоторых программ.

Кроме того, ключ -mt может выключать некоторые оптимизации, опасные при многопоточном исполнении. Поэтому если компилятор поддерживает ключ -mt, рекомендуется его использовать как при компиляции, так и при сборке многопоточных программ. Компилятор GCC не поддерживает ключ `-mt`, вместо этого рекомендуется использовать ключ -threads или -pthread на тех платформах, где эти ключи поддерживаются. GCC, входящий в поставку Solaris 10, поддерживает ключи `-threads` и `-pthread` ; GCC из поставки Debian Sarge поддерживает только ключ `-pthread`.

Ниже приводится пример кода, который позволяет протестировать ваш компилятор и проверить наличие типичных препроцессорных символов, используемых в include-файлах стандартной библиотеки языка C для проверки того, однопоточная или многопоточная программа сейчас компилируется. `_LIBC_REENTRANT` используется в Linux, `_REENTRANT` в Solaris. На других платформах могут использоваться другие символы. Попробуйте собрать эту программу с разными ключами компилятора и проверить результат. Полезно также поискать соответствующие символы в файлах каталога `/usr/include` и посмотреть, какие именно конструкции они контролируют.

```
int main() {
 #if defined(_LIBC_REENTRANT)
```

```
  printf("_LIBC_REENTRANT defined as %d\n", _LIBC_REENTRANT);
 #endif

 #if defined(_REENTRANT)
  printf("_REENTRANT defined as %d\n", _REENTRANT);
 #endif

 #if defined(_TS_ERRNO)
  printf("_TS_ERRNO defined as %d\n", _TS_ERRNO);
 #endif

 #if defined(_POSIX_C_SOURCE)
  printf("_POSIX_C_SOURCE defined as %d\n", _POSIX_C_SOURCE);
 #endif
}
```

Пример 2.1.

5. Программы для экспериментов

Получите вывод препроцессора для программы примера 2.2 с различными ключами компиляции. Вывод препроцессора у большинства компиляторов C (в том числе у компиляторов Sun Studio 11 и GCC) генерируется ключом `-E` и выдается в `stdout`. Для того, чтобы перенаправить вывод препроцессора в файл, используйте переназначение ввода-вывода. Можно также использовать для просмотра вывода компилятора фильтры `more(1)` или `less(1)`.

Напоминаю, что переменная errno хранит код ошибки последнего неудачно завершенного системного вызова. Код ошибки `EMFILE` означает исчерпание лимита дескрипторов файлов на процесс. Таким образом, функция `open_with_wait` пытается открыть файл и, если лимит дескрипторов исчерпан, блокирует нить в надежде, что какая-то другая нить освободит дескриптор. Функции `pthread_cond_wait(3C)`, `pthread_cond_signal(3C)`, `pthread_mutex_lock(3C)`, `pthread_mutex_unlock(3C)` и используемые ими типы данных изучаются далее в нашем курсе.

Посмотрите, в какой код превращается обращение к переменной `errno`.

Посмотрите, как этот код зависит от используемых ключей компиляции. Найдите в файле `/usr/include/errno.h` макроопределения, ответственные за эту замену. От каких предопределенных символов препроцессора они зависят?

Подумайте, к чему привело бы в многопоточной программе обращение к переменной errno как к обычной переменной. Можно ли, как это рекомендуется в некоторых старых руководствах по Unix, описывать переменную errno как extern int errno, или следует обязательно использовать определение из файла `/usr/include/errno.h`?

```
pthread_cond_t fileopen_cond=PTHREAD_COND_INITIALIZER;
pthread_mutex_t fileopen_mutex=PTHREAD_MUTEX_INITIALIZER;

int open_with_wait(const char *pathname, int flags, mode_t mode) {
  int code;
  pthread_mutex_lock(&fileopen_mutex);
  do {
    code=open(pathname, flags, mode);
    if (code < 0 && errno==EMFILE) {
      pthread_cond_wait(&fileopen_cond, &fileopen_mutex);
    }
  } while (code < 0 && errno==EMFILE);
  pthread_mutex_unlock(&fileopen_mutex);
  return code;
}

int close_with_wakeup(int handle) {
  int code;
  code=close(handle);
  pthread_cond_signal(&fileopen_cond);
  return code;
}
```

Пример 2.2.

Создание и завершение нитей

В ходе этого раздела вы изучите: создание нитей с атрибутами по умолчанию, передачу параметров нити, завершение нити, ожидание завершения другой нити, принудительное завершение нити, обработку принудительного завершения нити.

Если темы там возникнут – сразу снять

Бить нельзя их, а вот не вникнут - разъяснять

В. Высоцкий

1. Создание нитей с атрибутами по умолчанию

В POSIX Thread API нить создается библиотечной функцией `pthread_create(3C)`.

Параметры этой функции:

- `pthread_t * thread` – Выходной параметр. Указатель на переменную, в которой при успешном завершении будет размещен идентификатор нити.
- `const pthread_attr_t * attr` – Входной параметр. Указатель на структуру, в которой заданы атрибуты нити (рассматривается на следующей лекции). Если этот указатель равен
- `NULL`, используются атрибуты по умолчанию.
- `void *(*start_routine)(void*)` – Входной параметр. Указатель на функцию, которая будет запущена во вновь созданной нити.
- `void *arg` – Входной параметр. Значение,которое будет передано в качестве параметра `start_routine`.

Возвращаемое значение

- `0` при успешном завершении
- Код ошибки при неудачном завершении

Большинство других функций POSIX Threads API используют аналогичное соглашение о кодах возврата. Если в нашем учебном пособии у функции не указано описание кода возврата, значит, что она возвращает 0 при успешном завершении и код ошибки при ошибке. В страницах man(1) всегда указывается точное описание кода возврата всех функций.

Коды ошибок

- Значения кодов ошибок определены в виде символов препроцессора в файле `errno.h`
- `EAGAIN` – системе не хватает ресурсов для создания нити. Возможно, не хватает памяти под стек, исчерпан архитектурный лимит на количество нитей в процессе (`PTHREAD_THREADS_MAX`) либо административное ограничение на количество нитей. Как и у остальных системных вызовов, код `EAGAIN` означает, что повторный вызов функции с теми же параметрами может не привести к ошибке.
- `EINVAL` – один из параметров имеет недопустимое значение. Например, указатель на `start_routine` указывает на страницу памяти, исполнение которой запрещено.
- `EPERM` – вы не имеете полномочий для исполнения нити с заданными атрибутами. Например, вы не можете установить заданные в структуре `attr` класс планирования или приоритет.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>

void * thread_body(void * param) {
  sleep(1);
  printf("Child\n");
}
/*
* creates thread with default attributes and no parameters
* sleeps enough time to ensure that thread will print his message
* (under reasonable conditions)
```

```
*/
int main(int argc, char *argv[]) {
  pthread_t thread;
  int code;
  int i;
  code=pthread_create(&thread, NULL, thread_body, NULL);
  if (code!=0) {
    char buf[256];
    strerror_r(code, buf, sizeof buf);
    fprintf(stderr, "%s: creating thread: %s\n", argv[0], buf);
    exit(1);
  }
  sleep(5);
  return (EXIT_SUCCESS);
}
```

Пример 3.1.

Важное замечание

Параметр функции нити описан как `void *`, но библиотека никогда не пытается обращаться к нему как к указателю. Поэтому этот указатель можно использовать либо как ссылку на структуру (блок параметров нити), либо для передачи скалярного значения.

При передаче структур данных в качестве параметра нужно проявлять осторожность.

Во первых, не следует передавать структуры данных, размещенные в стеке родительской нити, то есть переменные с классом памяти `auto` и блоки памяти, размещенные при помощи `alloca(3C)`. Действительно, если ваша нить вернет управление из текущей функции или вообще завершится до того, как созданная нить начнет работать с блоком параметров, получится, что вы передали в качестве параметра висячую ссылку.

Во вторых, при передаче структур данных, размещенных при помощи `malloc(3C)` или оператора `new` языка C++, необходимо решить вопрос о том, кто будет освобождать эту структуру. Если структура не будет освобождена при помощи `free(3C)` или оператора `delete`,

это приведет к утечке памяти. Существует несколько решений этого вопроса, приемлемых в разных ситуациях.

Одно из решений состоит в том, что родитель размещает структуру, а созданная нить ее освобождает. Но обычно это считается дурным тоном при программировании на C/C++ (хорошим тоном считается, чтобы вызываемая функция не знала, каким образом выделена память под переданные ей параметры).

Другое решение состоит в том, что родитель размещает структуру, дожидается завершения потомка при помощи вызова `pthread_join(3C)` и освобождает структуру. В рамках этого подхода можно передавать как структуры данных, созданные при помощи `malloc(3C)`, так и структуры данных, размещенные в стеке родителя. Проблема этого подхода состоит в том, что если родитель будет принудительно завершен при помощи `pthread_cancel(3C)`, он может не дождаться завершения своих потомков, а это приведет либо к утечке памяти, либо к висячим ссылкам.

В программах с небольшим количеством нитей часто передают в качестве параметра указатели на статические переменные, но в больших программах с большим числом нитей это неприемлемо.

2. Завершение нити

Для завершения нити используется функция `pthread_exit(3C)`. Эта функция всегда завершается успешно и принимает единственный параметр, указатель на код возврата типа `void *`. Как и в случае с `pthread_create(3C)`, библиотека никогда не пытается обращаться к значению этого параметра как к указателю, поэтому его можно использовать для передачи скалярных значений.

Другой способ завершения нити – это возврат управления из функции `start_routine` при помощи оператора `return`. Поскольку функция `start_routine` должна возвращать тип `void *`, все ее операторы `return` должны возвращать значение. Этот способ практически полностью эквивалентен вызову `pthread_exit(3C)` с тем же самым значением.

Примечание

При компиляции программ на C++ некоторыми версиями компиляторов (наиболее известны проблемы такого рода с GCC, но проблемы есть и у некоторых коммерческих компиляторов), при завершении нити вызовом `pthread_exit(3C)` не вызываются деструкторы локальных переменных. Информацию об этом не всегда удается найти в документации. В действительности это определяется не столько версией компилятора, сколько опциями при сборке компилятора и особенностями C runtime/libpthread. Так, компилятор Sun Studio 11 вызывает деструкторы, а GCC 3.4.3, входящий в поставку Solaris 10 – не вызывает.

GCC 2.95.4 в Debian Woody не вызывает деструкторы, GCC 3.3.5 в Debian Sarge – вызывает.

Intel C++ Compiler 7.1 при работе на Debian Woody не вызывает деструкторы.

Протестировать взаимодействие POSIX Thread API и вашего компилятора C++ можно при помощи программы из примера 3.2.

```
#include <pthread.h>
#include <unistd.h>
#include <iostream>

class TestClass {
  public:
  int value;
  TestClass(int parameter) {
    std::cout << "Constructed " << parameter << "\n";
    value=parameter;
  }
  ~TestClass() {
    std::cout << "destructed " << value << "\n";
  }
};

extern "C" {
  void *body_forexit(void * param) {
```

```
    TestClass a(1);
    sleep(1);
    pthread_exit((void *)a.value);
    return (void *)a.value;
  }

  void *body_forreturn(void * param) {
    TestClass b(2);
    sleep(2);
    return (void *)b.value;
  }
} // extern "C"

int main() {
  pthread_t thread1, thread2;

  pthread_create(&thread1, NULL, body_forexit, NULL);
  pthread_detach(thread1);
  pthread_create(&thread2, NULL, body_forreturn, NULL);
  pthread_detach(thread2);
  pthread_exit(NULL);
  return 0;
}
```

Пример 3.2.

Примечание 2

Завершение процесса системным вызовом `exit(2)` или возвратом из функции `main` приводит к завершению всех нитей процесса. Это поведение описано в стандарте POSIX, поэтому ОС, которые ведут себя иначе (например, старые версии Linux), не соответствуют стандарту. Если вы хотите, чтобы нити вашей программы продолжали исполнение после завершения `main`, следует завершать `main` при помощи вызова `pthread_exit(3C)`.

Поведение деструкторов локальных переменных в нитях при завершении процесса при помощи `exit(2)` не описано в стандарте. На практике они не вызываются даже для переменных, описанных в

основной нити программы, и даже в однопоточных программах (проверялось в Solaris и Linux на всех доступных версиях компиляторов).

В действительности, все еще хуже – при выходе по `exit(2)` начинают вызываться деструкторы статических и глобальных переменных, но нити продолжают работу – в некоторых случаях это может приводить к аварийному завершению программы. Так, если в программе примера 3.2 в строке 42 закомментировать `pthread_exit`, то программа, собранная Sun Studio 11, с высокой вероятностью завершается по Segmentation Fault (попытка вывода в `std::cout` после того, как деструктор `std::cout` уже отработал). Поэтому если вы хотите экстренно завершить многопоточную программу, написанную на C++, используйте `_exit(2)`. Если вам необходимо обеспечить вызов деструкторов локальных переменных, следует использовать исключенияC++.

Хотя для ряда платформ обеспечивается удовлетворительное взаимодействие POSIX Thread library со средой исполнения C++, написание многоплатформенных многопоточных приложений на C++ может оказаться сложной задачей. Во многих случаях для написания переносимых многопоточных приложений на C++ лучше использовать не непосредственно POSIX Threads, a BOOST threads.

К передаче выходного значения нити относятся все те же самые соображения, что и к передаче параметров нити. Хотя выходное значение нити описано как `void *`, библиотека никогда не обращается к нему как к указателю, поэтому в качестве значения можно возвращать ноль или другие скалярные значения.

При передаче указателей возникают те же проблемы, что и при передаче параметров – опасность возникновения висячих ссылок и опасность утечки памяти. Ни в коем случае нельзя возвращать указатели на локальные переменные и другие объекты, созданные в стеке нити, потому что нить, получающая возвращаемое значение функцией `pthread_join(3C)` (рассматривается ниже), получает доступ к значению лишь после того, как стек нити и другие ресурсы, связанные с нитью, в том числе `thread local data`, уже уничтожены.

При использовании для размещения возвращаемого значения

malloc(3C) возникает опасность утечки памяти. Действительно, если для размещения значения использовать malloc(3C), то нить, получающая наше значение, должна его освободить. Но если эта нить получит сигнал принудительного завершения (pthread_cancel(3C)), то она не дождется нашего завершения и не освободит наше значение.

Универсального решения этой проблемы не существует. В программах с фиксированным или ограниченным числом нитей часто используют для передачи параметров и выходных значений указатели на статические переменные. В программах с переменным числом нитей обычно используют структуру, когда одна нить создает все остальные нити, размещает для них блоки переменных состояния, дожидается завершения каждой из нитей и уничтожает или переиспользует эти блоки. При этом, разумеется, логично, чтобы нить получала указатель на свой блок переменных состояния как параметр, работала с ним в течение всего срока жизни и возвращала этот же блок в качестве параметра pthread_exit(3C). Но эта архитектура пригодна не для всех приложений.

3. Ожидание завершения нити

Для ожидания завершения нити и получения ее кода возврата используется библиотечная функция pthread_join(3C). Эта функция имеет два параметра, идентификатор нити типа pthread_t, и указатель на переменную типа void *, в которой размещается значение кода возврата. Если в качестве второго параметра передать нулевой указатель, код возврата игнорируется.

Если требуемая нить еще не завершилась, то нить, сделавшая вызов pthread_join(3C), блокируется. Если такой нити (уже) не существует, pthread_join(3C) возвращает ошибку ESRCH.

Когда нить завершается, то связанные с ней ресурсы существуют до того момента, пока какаято другая нить не вызовет pthread_join(3C). Однако к тому моменту, когда pthread_join завершается, все ресурсы, занятые нитью (стек, thread local data, дескриптор нити) уничтожаются.

В отличие от процессов Unix, где системный вызов `wait(2)` может использовать только родитель по отношению к потомкам, любая нить может ждать завершения любой другой нити того же процесса. Если несколько нитей ждут завершения одной и той же нити, которая еще не завершилась, все эти нити блокируются. Однако при завершении нити одна из ожидавших нитей получает код возврата, а остальные ошибку `ESRCH`.

Если нить пытается ожидать сама себя, она получает ошибку `EDEADLK`.

Еще одна важная функция, связанная с ожиданием завершения нити – это функция `pthread_detach(3C)`. Эта функция указывает, что все ресурсы, связанные с нитью, необходимо уничтожать сразу после завершения этой нити. При этом уничтожается и код возврата такой нити – при попытке сделать `pthread_join(3C)` на нить, над которой перед этим сделали `pthread_detach(3C)`, возвращается код ошибки `EINVAL`.

В руководстве по `pthread_detach(3C)` в системе Solaris 10 сказано, что главное применение `pthread_detach(3C)` – это ситуация, когда родитель, ожидавший завершения дочерней нити, получает `pthread_cancel(3C)`. В действительности, существуют и другие применения "отсоединенных" нитей.

Не обязательно делать `pthread_detach(3C)` на уже запущенную нить; в атрибутах нити (`pthread_attr_t`) можно указать, что нить следует запускать в уже "отсоединенном" состоянии.

Это рассматривается в следующей лекции.

Разумеется, нить, над которой сделали `pthread_detach(3C)`, не должна выделять память под возвращаемое значение при помощи `malloc(3C)` – ведь никто не сможет освободить эту память и это приведет к утечке памяти.

Рекомендованного стандартом способа проверить собственную "отсоединенность" нет. Из предыдущего описания можно предположить, что для проверки "отсоединенности" можно было бы использовать код возврата `pthread_join(3C)` для собственного

идентификатора нитей – для "отсоединенных" нитей это должен быть `EINVAL`, а для "неотсоединенных" – `EDEADLK`.

Для Solaris 10 и Linux 2.6 это действительно так (во всяком случае для Debian Sarge), однако в Linux 2.4 `ptread_join(pthread_self(), NULL)` всегда возвращает `EDEADLK`. Как ведет себя `pthread_join` на вашей системе, можно проверить при помощи программы примера 3.3.

Кроме нестандартности, данный способ не гарантирует, что вас не "отсоединят" уже после того, как вы осуществили проверку. То есть надежда на результат такой проверки может привести к ошибке соревнования (race condition). Поэтому применять такую проверку в реальных приложениях не следует.

```
#include <pthread.h>
#include <errno.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

/* !!! NEVER USE THIS IN PRODUCTION ENVIRONMENT!!!! */
/* Behavior of this code implementation-dependent
 * using this code "as is" will result in race condition */

void * thread_body(void *unused) {
  pthread_t self;
  int code;
  self=pthread_self();

  code=pthread_join(self, NULL);

  printf("Joining self before detach, error code=%d, message %s\n",
        code, strerror(code));
  /* NEVER USE THIS CODE IN PRODUCTION ENVIRONMENT */
  pthread_detach(self);
  code=pthread_join(self, NULL);
  printf("Joining self after detach, error code=%d, message %s\n",
        code, strerror(code));
```

```
  return NULL;
}

int main() {
  pthread_t thread;
  pthread_create(&thread, NULL, &thread_body, NULL);
  pthread_exit(NULL);
  return 0;
}
```

Пример 3.3.

Поскольку стандартного способа проверить "отсоединенность" нити не существует, при разработке архитектуры приложения не следует полагаться на то, что способ этой проверки вообще доступен.

4. Принудительное завершение нити (прерывание)

Библиотечная функция `pthread_cancel(3C)` принудительно завершает нить. В зависимости от свойств нити и некоторых других обстоятельств, нить может продолжать исполнение некоторое время после вызова `pthread_cancel(3C)`.

Нить может установить одну или несколько функций-обработчиков, которые будут вызваны при попытке прервать исполнение этой нити. Такие обработчики могли бы освободить дополнительные ресурсы, занятые нитью, например блоки динамической памяти или мутексы, или привести в согласованноесостояние разделяемые структуры данных.

Момент, в который нить получает сообщение о попытке прервать ее исполнение функцией `pthread_cancel(3C)` контролируется атрибутами нити, известными как `cancel state` и `cancel type`.

`Cancel state` (состояние прерывания) определяет, разрешено ли прерывание нити как таковое. Т.е. этот атрибут может иметь два значения – разрешено или запрещено. Если прерывание разрешено, нить немедленно получает сообщение о попытке ее прервать (хотя, в зависимости от `cancel type`, может отреагировать на это

сообщение лишь через некоторое время). Если прерывание запрещено, попытки прерывания нити накапливаются. После того, как прерывания все-таки разрешат, нить получит сигналы о накопившихся попытках.

Переключение состояния прерывания осуществляется функцией `pthread_setcancelstate(3C)`.

Первый параметр этой функции входной и может принимать значения `PTHREAD_CANCEL_ENABLE` (прерывание разрешено) и `PTHREAD_CANCEL_DISABLE` (прерывание запрещено). Эти значения – препроцессорные макроопределения, содержащиеся в файле `pthread.h`. Вызов функции с другими значениями первого параметра приведет к ошибке `EINVAL`. Второй параметр функции – выходной, содержит указатель на переменную, в которой будет размещено старое значение типа прерывания. В качестве этого указателя можно передать `NULL`, в этом случае старое значение состояния будет потеряно. По умолчанию, нить создается с разрешенными прерываниями.

Cancel type (тип прерывания) определяет, в какие моменты нить проверяет сообщения о прерываниях. Этот атрибут может принимать два значения – `PTHREAD_CANCEL_DEFERRED` (отложенное прерывание) и `PTHREAD_CANCEL_ASYNCHRONOUS` (асинхронное прерывание). По умолчанию, нить создается с отложенным типом прерываний. Что означает каждое из возможных значений этого атрибута, описывается далее в этом разделе.

Установка типа прерывания осуществляется функцией `pthread_setcanceltype(3C)`. Схема передачи параметров этой функции аналогична `pthread_setcancelstate(3C)`.

Тип и состояние прерывания могут быть заданы в момент создания нити при помощи установки соответствующих полей в структуре `pthread_attr_t`. Обратите внимание, что оба эти атрибута задаются либо в момент создания нити, либо самой нитью. Внешними по отношению к нити средствами их изменить невозможно. Поэтому проверка значений этих атрибутов в определенные моменты времени не может приводить к ошибкам соревнования.

Асинхронное прерывание означает, что библиотека прерывает нить как

можно скорее (хотя во многих ситуациях не удается гарантировать, чтобы это происходило точно в тот момент, когда другая нить вызвала `pthread_cancel(3C)`). Асинхронное прерывание требует тщательного анализа всех возможных моментов, когда оно может произойти, и обработки всех ситуаций, связанных с прерываниями в неудачные моменты. Так, если прерывание произойдет во время работы с библиотекой, которая не считается *`thread-safe`*, внутренние данные этой библиотеки могут остаться в несогласованном состоянии.

Для корректной работы большинства библиотек в таких условиях простой поддержки многопоточности не достаточно. В страницах системного руководства Solaris уровень поддержки многопоточности описывается несколькими типами атрибутов, которые рассматриваются в лекции 5; безопасность библиотеки или функции при использовании асинронных прерываний описывается атрибутом MT-Level. У библиотеки, которая безопасна для применения в таких условиях, этот атрибут имеет значение Asynchronous-Cancel-Safe. Для всех библиотек и функций, для которых этот атрибут явным образом не указан на соответствующей странице системного руководства, следует предполагать, что они небезопасны для использования в режиме асинхронных прерываний.

Отложенное прерывание означает, что нить получает сообщение о прерывании лишь в определенные моменты, известные как точки прерывания (cancellation point). Эти точки, в свою очередь, делятся на две категории – явные и неявные. Явные точки прерывания – это вызовы функции `pthread_testcancel(3C)`. Неявные точки прерывания – это вызовы следующих функций и системных вызовов:

aio_suspend(3RT), close(2), creat(2), getmsg(2), getpmsg(2), lockf(3C), mq_rec
mq_send(3RT), msgrcv(2), msgsnd(2), msync(3C), nanosleep(3RT), open(2), pa
pread(2), pthread_cond_timedwait(3C), pthread_cond_wait(3C), pthread_join(3
pthread_testcancel(3C), putmsg(2), putpmsg(2), pwrite(2), read(2), readv(2), se
sem_wait(3RT), sigpause(3C), sigwaitinfo(3RT), sigsuspend(2), sigtimedwait(3R
sleep(3C), sync(2), system(3C), tcdrain(3C), usleep(3C), wait(3C), waitid(2), w
write(2), writev(2), fcntl(2) (с командой F_SETLKW).

Список приведен для Solaris 10. Получить полный список неявных точек прерывания для вашей версии ОС можно получить на странице

руководства `cancellation(5)`. Несколько забегая вперед, необходимо отметить, что функция `pthread_mutex_lock(3C)` не является неявной точкой прерывания, и в соответствии с требованиями стандарта POSIX, не должна являться таковой.

Стандарт POSIX допускает наличие неявных точек прерывания в некоторых других системных вызовах и библиотечных функциях, которые перечислены на странице ссылка: http://www.opengroup.org/onlinepubs/007908799/xsh/threads.html - http://www.opengroup.org/onlinepubs/007908799/xsh/threads.html. Слушатели, знакомые с реализацией стандартных библиотек языка C должны заметить, что это преимущественно функции, которые содержат или могут содержать вызовы неявных точек прерывания, перечисленных выше. В частности, это большинство функций буферизованного ввода-вывода (`printf(3C)`, `fread(3C)`, `fwrite(3C)`).

Однако реализации POSIX Thread Library в Solaris 10 и Linux 2.6 не содержат неявных точек прерывания ни в одной из этих функций. Во всяком случае, в Solaris 10 список функций, содержащих неявные точки прерывания, приведенный на странице руководства `cancellation(5)`, является исчерпывающим.

Слушателям, интересующимся вопросом, как удалось реализовать `printf(3C)` или `fwrite(3C)` без обращения к точке прерывания в теле `write(2)`, я рекомендую обратиться к исходным текстам библиотеки языка C, размещенным на сайте ссылка: http://www.opensolaris.org - http://www.opensolaris.org, или просто пройти в пошаговом отладчике код библиотечной функции `fwrite(3C)`. Для этого предварительно рекомендуется выключить буферизацию функцией `setvbuf(3C)`, тогда каждый вызов `fwrite` будет вызывать соответствующий системный вызов лишь после небольшого количества проверок, и будет легче понять логику кода.

Более сложный вопрос – зачем это было сделано. Ответ, скорее всего, состоит в том, что функции буферизованного ввода-вывода – не просто "обертки" над соответствующими системными вызовами, они осуществляют определенные (и иногда довольно сложные) операции над структурой данных, которая, собственно, и является буфером ввода-вывода. Прерывание этих библиотечных функций в неудачный момент

либо приводило бы к риску оставить буфер ввода-вывода в несогласованном состоянии, либо требовало бы сложных вычислений для отслеживания и восстановления его согласованности.

6. Обработка прерывания нити

Нить может установить одну или несколько функций, которые будут вызываться при получении ей сообщения о прерываниии. Эти функции могут восстанавливать согласованность структур данных, с которыми нить работала до получения прерывания, освобождать мутексы и другие примитивы синхронизации, занятые нитью, освобождать динамическую память. Функция-обработчик прерывания устанавливается макрокомандой

<code>pthread_cleanup_push(3C)</code>,

которая определена в файле `/usr/include/pthread.h`. Эта макрокоманда внешне выглядит как вызов одноименной функции с двумя параметрами – указателем на функцию-обработчик и значением, которое следует передать этой функции в качестве параметра.

Функции-обработчики вызываются в стековом порядке (Last In First Out), т.е. в порядке, обратном тому, в котором они устанавливались, и из того контекста, в котором они устанавливались. Иными словами, в качестве параметров функций-обработчиков можно использовать указатели на локальные переменные, определенные в тех блоках, в которых стоял вызов соответствующего `<code>pthread_cleanup_push(3C)</code>`.

Каждый вызов `<code>pthread_cleanup_push(3C)</code>` должен сопровождаться соответствующим вызовом парной макрокоманды `pthread_cleanup_pop(3C)`. Эта макрокоманда должна вызываться в пределах того же блока, в котором была вызвана соответствующая `<code>pthread_cleanup_push(3C)</code>`.

Нарушение этого требования приведет к ошибке компиляции. В действительности, макрокоманды `pthread_cleanup_push/pop` содержат, соответственно, открывающую и закрывающую фигурную скобки. Это не является требованием стандарта POSIX, но поможет

понять, к каким именно ошибкам компиляции может привести их неправильное использование.

Макроопределение `pthread_cleanup_pop(3C)` содержит параметр, который указывает, следует ли вызвать обработчик при его выталкивании из стека или нет. Если обработчик осуществляет какие-то сложные неатомарные операции, рекомендуется его вызывать, а не дублировать соответствующий код в теле основной функции нити.

Не следует использовать в функциях-обработчиках такие функции, как `lonjmp(3C)/siglongjmp(3C)` или оператор `throw` языка C++, это может привести, и в большинстве реализаций действительно приведет, к нарушению порядка вызова других обработчиков или даже к разрушению стека и аварийному завершению программы. В большинстве реализаций другие обработчики просто не будут вызваны.

Атрибуты нитей и управление нитями

В ходе этого раздела вы изучите: создание нитей с заданными атрибутами, управление атрибутами нитей, приоритеты нитей и классы планирования, вспомогательные функции для управления нитями , мониторинг поведения многопоточной программы, начальные сведения об отладке многопоточных программ.

Так встань у реки, смотри как течет река;

Ее не поймать ни в сеть, ни рукой.

Она безымянна, ведь имя есть лишь у ее берегов;

Прими свое имя и стань рекой.

Б. Гребенщиков

Атрибуты нитей

При создании нити можно указать блок атрибутов нити при помощи второго параметра функции `pthread_create(3C)`. Этот блок представляет собой структуру `pthread_attr_t`.

Стандарт POSIX требует рассматривать эту структуру как непрозрачный тип и использовать для изменения отдельных атрибутов функции просмотра и установки отдельных атрибутов. Для инициализации `pthread_attr_t` следует использовать функцию `pthread_attr_init(3C)`. Эта функция имеет единственныйпараметр – `pthread_attr_t *attr`. Она устанавливает атрибуты в соответствии с табл. 4.1.

Таблица 4.1.

Атрибут	Значение по умолчанию	Объяснение
		Нить использует процессорное врем выделяемое процес В Solaris 9 и

scope	PTHREAD_SCOPE_PROCESS	последующих верси Solaris этот парамет не имеет практического значения
detachstate	PTHREAD_CREATE_JOINABLE	Нити создаются присоединенными (для освобождения ресурсов после завершения нити необходимо вызват pthread_join( ).
stackaddr	NULL	Стек нити размеща системой
stacksize	0	Стек нити имеет размер, определяем системой
priority	0	Нить имеет приори 0
inheritsched	PTHREAD_EXPLICIT_SCHED	Нить не наследует приоритет у родите
schedpolicy	SCHED_OTHER	Нить использует фиксированные приоритеты, задаваемые ОС. Используется вытесняющая многозадачность (н исполняется, пока н будет вытеснена другой нитью или н заблокируется на примитиве синхронизации)

Кроме этих атрибутов, pthread_attr_t содержит некоторые другие атрибуты. Некоторые из этих атрибутов игнорируются Solaris 10, и введены для совместимости с другими реализациями POSIX Thread API.

Для изменения значений атрибутов необходимо использовать специальные функции. Для каждого атрибута определены две функции, pthread_attr_set%ATTRNAME% и pthread_attr_get%ATTRNAME%.

При создании нити используются те значения атрибутов, которые были заданы к моменту вызова pthread_create(3C). Изменения в pthread_attr_t, внесенные уже после запуска нити, игнорируются системой. Благодаря этому один и тот же блок атрибутов нити можно использовать для создания нескольких разных нитей с разными атрибутами.

Некоторые из атрибутов, например detachstate и priority, могут быть изменены у уже работающей нити. Для этого используются отдельные функции. Так, detachstate в атрибутах нити изменяется функцией pthread_attr_setdetachstate(3C), а у исполняющейся нити – функцией pthread_detach(3C). Функция pthread_detach(3C) обсуждалась на предыдущей лекции. Pthread_detach (3C) изменяет значение атрибута с JOINABLE на DETACHED ; способа изменить этот атрибут с DETACHED на JOINABLE у работающей нити не существует.

Многие из атрибутов, например размер или местоположение стека, не могут быть изменены у работающей нити.

Перед уничтожением структуры pthread_attr_t необходимо вызвать функцию pthread_attr_destroy(3C).

Ниже приводится список атрибутов нити с их допустимыми значениями и кратким описанием семантики.

Scope (область действия). Допустимые значения: PTHREAD_SCOPE_SYSTEM и PTHREAD_SCOPE_PROCESS. Значение PTHREAD_SCOPE_SYSTEM означает, что нить планируется системным планировщиком и соревнуется за системные ресурсы с

другими процессами. Значение PTHREAD_SCOPE_PROCESS означает, что нить планируется пользовательским планировщиком и, с точки зрения системы, считается частью своего процесса. В системах с гибридным планировщиком нити с областью действия SYSTEM соответствует так называемым "привязанным" (bound) нитям, которые привязаны к определенному LWP. В Solaris 9 и 10 все нити исполняются в собственныхLWP, поэтому установка атрибута scope не имеет практического значения (возможно, правильнее было бы указать PTHREAD_SCOPE_SYSTEM как значение по умолчанию). Detachstate (присоединение/отсоединение). Допустимые значения: PTHREAD_CREATE_JOINABLE и PTHREAD_CREATE_DETACHED. У нитей в состоянии DETACHED ресурсы (стек, Thread Local Data) освобождаются сразу после завершения нити.

У нитей в состоянии JOINABLE ресурсы освобождаются после того, как другая нить вызовет pthread_join(3C).

Stacksize (размер стека). Допустимые значения – 0 или размер стека в байтах. 0 означает, что система сама определяет размер стека. В Solaris на 32-битных платформах под стек выделяется 1 мегабайт памяти, на 64-битных – 2 мегабайта. Запрещено выделять под стек меньше памяти, чем PTHREAD_STACK_MIN. Эта константа определена в <limits.h> и в <pthread.h>. Попытка указать стек меньшего размера может привести к аварийному завершению вашей программы, скорее всего(но не обязательно) по SIGSEGV.

Эксперименты показывают, что простые программы могут успешно исполняться со стеками, значительно меньшими, чем PTHREAD_STACK_MIN, но, разумеется, никто не гарантирует, что они смогут исполняться таким образом при всех возможных сочетаниях обстоятельств.

При разработке реальных программ необходимо иметь в виду, что значение PTHREAD_STACK_MIN учитывает только потребности POSIX Thread Library, но не потребности вашего кода и не потребности библиотек, которые ваш код может вызывать. В рамках нашего курса мы не изучаем средств, при помощи которых можно определить реальные потребности вашей программы в памяти под стек.

`Stackaddr` (адрес стека). Допустимые значения – `NULL` или указатель на область памяти, достаточную для размещения стека нити. `NULL` означает, что система должна разместить область под стек при создании нити. При выделении области под стекнеобходимо проявлять осторожность. Наиболее важный из практических аспектов – что если вы выделяли память под стек сами, то даже у отсоединенных нитей система не будет освобождать эту память (потому что система, вообще говоря, незнает, каким способом вы выделяли эту память). Разумеется, освобождать память из-под стека можно только после успешного завершения `pthread_join(3C)`. Освобождение памяти из-под стека отсоединенных нитей представляет собой трудноразрешимую на практике проблему.

Как правило, не рекомендуетсяразмещать память под стек самостоятельно. В большинстве случаев разумнее доверить эту работу системе.

`Concurrency` (степень параллелизма). Допустимые значения – положительные целые числа.

В системах с гибридным планировщиком значение атрибута с определенными допущениями соответствует количеству LWP, создаваемых системой для вашего процесса.

В Solaris 9 и 10 установка `concurrency` игнорируется системой. Сам атрибут сохраняется для совместимости со стандартом POSIX и старыми приложениями. `Schedpolicy` (политика планирования). Допустимые значения – `SCHED_FIFO`, `SCHED_RR` и `SCHED_OTHER`. Политики `SCHED_FIFO` и `SCHED_RR` используются процессами реального времени.

FIFO (First-In First-Out, первый вошел, первый вышел) соответствует планированию в порядке линейной очереди; нити получают управление и отдают его только когда остановятся на примитиве синхронизации или вызовут `sched_yield(3RT)`.

RR (Round-Robin, кольцевая очередь) соответствует планированию с квантами времени, когда нити исполняются либо до момента явной отдачи управления, либо до истечения кванта времени. При истечении

кванта временинить устанавливается в конец очереди. В Solaris процессы и нити реального времени может исполнять только суперпользователь с uid==0. Установка политик `SCHED_FIFO` и `SCHED_RR` другими пользователями не является ошибкой, однако нить по прежнему исполняетсяс классом планирования, унаследованным от родительского процесса.

`SCHED_OTHER` соответствует системной политике планирования. В Solaris нити наследуют политику планирования и приоритет своего процесса. При помощи системного вызова `priocntl(2)` можно назначать приоритет и класс планирования для как для отдельных LWP, так и для процессов в целом и для групп процессов. Этот способ управления политиками планирования обеспечивает гораздо большую гибкость, чем POSIX Thread API, но он нестандартен. Так, в Linux для управления приоритетами используются вызовы `get/setpriority(2)`, а понятия класса планирования в Linux вообще нет. Наиболее портабельный API для управления приоритетами в системах семейства Unix – это системный вызов `nice(2)`, но этот API отличается негибкостью и также не позволяет управлять классами планирования.

Все системы Unix SVR4 поддерживают два класса планирования – TS (Time Share, разделяемое время) и RT (Real Time, реальное время).

Time Share – система оптимизирует среднее время реакции на внешние события. Используется простая динамическая приоритизация, при которой приоритет процесса складывается из двух компонентов: базового приоритета (nice level) и штрафа за исчерпание кванта времени. Базовый приоритет задается системным вызовом `nice(2)`, чем больше значение `nice`, тем ниже приоритет. Штраф накладывается на нити, которые были сняты системой по истечению кванта времени, и увеличивается с каждым последующим таким снятием. Если нить исполняет блокирующийся системный вызов, штрафснимается. На практике это приводит к тому, что у процессов и LWP, исполняющих блокирующиеся системные вызовы (такие процессы вносят наибольший вклад во время реакции системы), относительный приоритет повышается.

RealTime – используется статическая приоритизация. LWP реального

времени исполняется до тех пор, пока не будет вытеснен LWP с более высоким приоритетом (это может произойти либо в результате выхода другого LWP из блокирующегося системного вызова, либо в результате изменения приоритета нашего LWP), либо пока сам не отдаст управление, исполнив блокирующийся системный вызов или `sched_yield(3RT)`. LWP реального времени имеют более высокий приоритет, чем любые процессы разделенного времени, а также более высокий приоритет, чем большинство нитей ядра. Только суперпользователь имеет право запускать LWP и процессы реального времени. Solaris также поддерживает дополнительные классы планирования – FX (Fixed Priority, фиксированные приоритеты) и FSS (Fair Share Scheduling, справедливое планирование). Fair Share Scheduling – сложная схема динамической приоритизации, обеспечивающая определенным группам процессов гарантированную долю процессорного времени.

Используется при планировании заданий в рамках проектов и контейнеров Solaris. Модули ядра, реализующие классы планирования FSS и FX, входят в стандартную поставку Solaris 10, однако администратор системы может не загружать эти модули, тогда соответствующие классы планирования будут недоступны. Документация по настройке системы настоятельно не рекомендует устанавливать классы планирования TS и FSS на одном процессоре, потому что процессы с этими классами планирования используют одни и те же уровни приоритетов; соревнование таких процессов за один процессор может приводить к труднопредсказуемым и, как правило, нежелательным явлениям. Как правило, рекомендуется устанавливать классы планирования FSS и TS на разных процессорах многопроцессорных компьютеров, или использовать только один из этих классов планирования.

`Inheritsched` (наследование класса планирования). Допустимыезначения – `PTHREAD_INHERIT_SCHED` и `PTHREAD_EXPLICIT_SCHED`. `INHERIT` (наследовать) означает, что нить наследует класс планирования и приоритет от родительской нити, `EXPLICIT` (задавать явно) означает, что класс планирования и приоритет вновь создаваемой нити должны быть указаны явно в атрибуте `schedparam`

`Schedparam` (параметры планирования). Допустимое значения – `struct sched_param`, определенная в <sched.h>. В соответствии со стандартом POSIX эта структура должна содержать поле `sched_priority`, соответствующее приоритету нити. В сочетании с политикой планирования `SCHED_OTHER` в Solaris 10 используются также поля `sched_nice` и `sched_nicelim`. `Guardsize` (размер сторожевой области). Допустимые значения – 0 или желаемый размер сторожевой области в байтах. Значение 0 указывает, что сторожевую область выделять не надо.

Сторожевая область (guard area) – это страницы или сегменты, защищенные от чтения и записи, размещаемые в начале области, выделяемой под стек. Если стек нити при своем росте достигает сторожевой области, программа завершается по `SIGSEGV`. Это используется для защиты от переполнения стека, которое может происходить из-за бесконечной рекурсии или злоупотребления большими структурами данных, размещаемыми в стеке. Некоторые языки программирования допускают размещение в стеке динамических структур данных. В C99 допускаются динамические массивы, во многих реализациях языка C память в стеке можно размещать при помощи нестандартной функции `alloca(3C)`.

Неконтролируемое переполнение стека приводит к неконтролируемому разрушению размещенных в памяти структур данных и непредсказуемым последствиям. Использование сторожевой области, во всяком случае, гарантирует, что `SIGSEGV` произойдет вскоре после собственно переполнения. Это обычно сильно упрощает отладку программы.

Размер сторожевой области следует выбирать исходя из размера структур данных, которые ваша программа размещает в стеке. Система имеет право округлять этот размер вверх до числа, кратного размеру страницы диспетчера памяти (этот размер можно получить системным вызовом `sysconf(2)` с параметром `PAGESIZE`). Однако `pthread_attr_getguardsize(3C)` возвращает значение `guardsize` без учета округления.

По умолчанию, система выделяет под сторожевую область одну страницу.

Дополнительные функции по управлению нитями

Для инициализации какого-либо динамического пакета в многопоточной программе рекомендуется использовать функцию `pthread_once(3C)`. Параметры этой функции:

- `pthread_once_t *flag` – флаговая переменная, которая контролирует, вызывалась функция или еще нет
- `void (*init_routine)(void)` – функция, которую следует вызвать.

`Pthread_once(3C)` гарантирует, что `init_routine` будет вызвана один раз. Если вы вызовете `pthread_once` с одной и той же флаговой переменной из двух разных нитей, то функция `init_routine` будет вызвана в одной нити и не будет вызвана в другой. Перед использованием `pthread_once` необходимо инициализировать флаговую переменную константой `PTHREAD_ONCE_INIT`. Результаты вызова `pthread_once` с неинициализированной флаговой переменной или с флаговой переменной с классом памяти `auto` непредсказуемы.

Сама по себе функция `pthread_once(3C)` не является точкой прерывания. Однако если `init_routine` содержит точки прерывания и нить будет прервана во время исполненияэтой функции, результат будет такой, как будто `init_routine` не вызывалась. Т.е. при вызове `pthread_once(3C)` из другой нити `init_routine` будет вызвана повторно. Для передачи управления между нитями можно использовать нестандартную функцию `sched_yield(3RT)`. Эта функция снимает текущую нить с процессора, но не позволяет контролировать, какой из других нитей этого или другого процесса будет передан процессор. Эта функция включена в курс потому, что без нее может быть сложно решить задачу 10. Однако этой функцией не следует злоупотреблять.

Если функциональность вашей программы зависит от расстановки в коде вызовов `sched_yield(3RT)`, то это ошибочная программа. Результат исполнения такой программы будет также зависеть от количества процессоров в системе, от распределения нитей вашей

программы между процессорами, а также от наличия в системе других активных нитей и процессов и от того, чем именно занимаются эти процессы.

Таким образом, `sched_yield(3RT)` можно использовать лишь для выполнения нефункциональных требований, скорее всего связанных с временемреакции на события. По видимому, именно из этих соображений в Solaris 10 эта функция включена в библиотеку `librt.so` (так называемую библиотеку реального времени), а не в основной API. При сборке программ, использующих `sched_yield(3RT)` необходимо подключать эту библиотеку ключом `-lrt`.

Функция `sched_yield(3RT)` нестандартна; в Linux аналогичная функция называется `pthread_yield(3PTHREAD)`, в других реализациях POSIX Thread API аналогичных функций может вообще не быть.

Приватные данные нити

Приватные или локальные данные нити (Thread Specific Data) – это переменные, создаваемые функцией `pthread_key_create(3C)`. Эти переменные имеют тип `void *` и идентифицируются ключами `pthread_key_t`. Для каждой нити, которая пытается обратиться к переменной, идентифицируемой определенным ключом, создается собственная копия значения соответствующей переменной.

Параметры функции `pthread_key_create(3C)`:

- `pthread_key_t *key` – ключ, идентифицирующий переменную локальных данных. Это непрозрачный тип, внутренняя структура которого зависит от реализации POSIXThread API.
- `void (*destructor)(void *)` – необязательная функция-деструктор.

При завершении нити, если значение деструктора отлично от нуля и если значение самой переменной отлично от нуля, будет вызван

деструктор, а в качестве параметра ему будет передано значение переменной. Например, если в качестве значения переменной используется блок памяти, выделенный при помощи `malloc(3C)`, в деструкторе следует освободить эту память при помощи `free(3C)`. Деструктор должен продемонстрировать свое завершение, установив значение переменной в `NULL`. Деструкторы вызываются в любом порядке, но если после отработки всех деструкторов значения некоторых переменных не равны нулю, деструкторы этих переменных будут вызваны повторно. Количество повторных вызовов деструкторов определяется константой `PTHREAD_DESTRUCTOR_ITERATIONS`.

Многократный вызов `pthread_key_create(3C)` с одной и той же переменной-ключом, вообще говоря, приводит к многократному созданию ключей, поэтому `pthread_key_create(3C)` следует вызывать либо до создания нитей, работающих с этим ключом, либо при помощи `pthread_once(3C)`.

Доступ к значению локальной переменной осуществляется функциями `pthread_setspecific(3C)` и `pthread_getspecific(3C)`. Если вызвать `get` до первого вызова `set` из той же нити, будет получено значение `NULL`.

Приватные данные нитей широко используются при переделке небезопасных при многопоточном исполнении программ в безопасные. Действительно, основной источник проблем с многопоточностью – это статические переменные, используемые внутри библиотеки. Преобразовав обращения к таким переменным в `thread-specific data`, мы в большинстве случаем можем получить корректно работающую версию библиотеки. Например, функция `strtok(3C)` при первом вызове разбивает строку-параметр на токены в соответствии с заданными разделителями и возвращает первый токен. При последующих вызовах она возвращает второй, третий и т.д. токены. Для этого она должна где-то хранить указатель на текущий токен; однопоточные версии стандартной библиотеки языка C обычно используют для этого статическую переменную. Разумеется, если в другой нити эта функция будет вызвана с другой начальной строкой, этот указатель будет перезаписан. Результат следующего вызова `strtok(3C)` в первой нити при этом будет сильно отличаться от того, что, скорее всего, ожидал программист, разрабатывавший программу

этой нити.

Если заменить статическую переменную на TSD – как это сделано в libc, входящей в поставку Solaris 10 – мы получим многопоточную версию `strtok(3C)`, которая ведет себя в соответствии с разумными ожиданиями. При этом большинство библиотек, использующих `strtok(3C)` разумным образом, также станут безопасны с точки зрения многопоточности (разумеется, при условии, что в них нет других проблем с многопоточностью).

Очевидно, что такой подход не является универсальным. Так, если подвергнуть такому преобразованию реализацию `malloc(3C)/free(3C)`, это приведет к тому, что у каждой нити будет свой пул памяти, и память, выделенную в одной нити, нельзя будет освобождать в другой нити. Такое поведение нельзя назвать соответствующим разумным ожиданиям.

Мониторинг и отладка

Использование ядерных нитей дает определенные преимущества при анализе поведения процесса. Поскольку все нити процесса соответствуют системным объектам, стандартные средства мониторинга системы могут показать нам, сколько у процесса нитей и чем они занимаются в данный момент.

В Linux нити процесса имеют собственный `pid(ProcessIdentifier)` и показываются как отдельные записи в таблице процессов, например в выводе команд `ps(1)` и `top(1)`.

В Solaris, по умолчанию процесс вместе со всеми своими нитями показывается как одна строка в выводе команд `ps(1)` и `prstat(1)`. У команды `ps(1)` некоторые опции, в частности параметр `nlwp` у опции –o, позволяют вывести количество LWP у процесса. У команды `prstat(1)` количество LWP выводится в последней колонке, через / после имени программы (см. пример 4.1).

```
PID USERNAME SIZE RSS STATE PRI NICE TIME CPU PROCESS/NLW
10294 vinokuro 123M 51M sleep 59 0 0:02:05 0.0% java/23
11430 fat 4720K 4192K cpu0 59 0 0:00:00 0.0% prstat/1
```

```
  744 root 1924K 1264K sleep 59 0 0:00:00 0.0% snmpdx/1
  646 root 3440K 1828K sleep 59 0 0:00:00 0.0% syslogd/13
  713 root 1680K 796K sleep 59 0 0:00:00 0.0% smcboot/1
  714 root 1680K 796K sleep 59 0 0:00:00 0.0% smcboot/1
  641 daemon 4216K 3700K sleep 59 0 0:00:01 0.0% nfsmapid/4
  635 root 1680K 1032K sleep 59 0 0:00:00 0.0% sac/1
  637 root 1788K 1212K sleep 59 0 0:00:00 0.0% ttymon/1
  634 root 1048K 720K sleep 59 0 0:00:00 0.0% utmpd/1
  639 root 3584K 2576K sleep 59 0 0:00:00 0.0% inetd/3
  619 daemon 2476K 1836K sleep 59 0 0:00:00 0.0% statd/1
  627 daemon 2040K 1452K sleep 59 0 0:00:00 0.0% lockd/2
  578 root 5248K 4068K sleep 59 0 0:00:43 0.0% nscd/33
  524 root 6172K 5340K sleep 59 0 0:00:08 0.0% svc.configd/12
  522 root 8684K 6448K sleep 59 0 0:00:30 0.0% svc.startd/12
  554 daemon 3908K 2332K sleep 59 0 0:00:01 0.0% kcfd/4
  625 root 2016K 1500K sleep 59 0 0:00:00 0.0% ypbind/1
  520 root 1992K 1228K sleep 59 0 0:00:00 0.0% init/1
  507 root 0K 0K sleep 60 -0:00:00 0.0% zsched/1


 5719 belenky 944K 600K sleep 59 0 0:00:00 0.0% a.out/1
Total: 66 processes, 164 lwps, load averages: 0.00, 0.00, 0.00
```

Пример 4.1. вывод команды prstat

Команда `ps -L` выводит информацию по каждому LWP отдельной строкой. Ключ `-L` совместим с большинством других ключей, например можно написать `ps -Lfu $USER`. Аналогично ведет себя команда `prstat -L`.

Информация обо всех LWP процесса доступна в каталоге этого процесса в псевдофайловой системе `/proc`. Эта информация доступна в подкаталоге `/proc/$PID/lwp/$LWPID`. Структуры размещенных в этом подкаталоге файлов описаны в `proc(4)`.

Ряд утилит, работающих с `proc(4)` позволяют получать информацию об LWP, например утилита `pstack(1)` выдает дамп стеков всех LWP

указанного процесса (см. пример 4.2). `Pstack(1)` приостанавливает исполнение процесса на время выдачи стека, но в целом не причиняет ему никакого вреда. Исследуя выдачу `pstack(1)` часто удается узнать много нового про свою программу и много забавного про реализацию библиотечных функций.

```
new AB
new A
new C
new BOLT

new A
^Z
[1]+ Stopped ./production_line
-bash-3.00$ ps


 PID TTY TIME CMD
11536 pts/9 0:00 ps
11410 pts/9 0:00 bash
11535 pts/9 0:00 producti


-bash-3.00$ pstack 11535
11535: ./production_line
-----------------lwp# 1 / thread# 1 --------------------


fef7d2a9 lwp_park (0, 0, 0)
fef733a0 sema_wait (8060dbc) + d
fefb565b sem_wait (8060dbc) + 27
08050a89 createWidget (8047d60, 8047cc4, feffa824, 5, 4, 3) + 19
08050b6c main (1, 8047d08, 8047d10) + ac
0805088a _start (1, 8047dc8, 0, 8047dda, 8047de5, 8047df5) + 7a


-----------------lwp# 2 / thread# 2 --------------------fef7d5e5
nanosleep (fee59fb0, fee59fb8)
08050936 createA (0) + 16
```

```
fef7cf3f _thr_setup (fee82400) + 4e
fef7d230 _lwp_start (fee82400, 0, 0, fee59ff8, fef7d230, fee82400)


-----------------lwp# 3 / thread# 3 -------------------fef7d5e5
nanosleep (fed4dfb0, fed4dfb8)
08050986 createB (0) + 16
fef7cf3f _thr_setup (fec30000) + 4e
fef7d230 _lwp_start (fec30000, 0, 0, fed4dff8, fef7d230, fec30000)


-----------------lwp# 4 / thread# 4 -------------------fef7d5e5
nanosleep (fec2dfb0, fec2dfb8)
080509d6 createC (0) + 16
fef7cf3f _thr_setup (fec30400) + 4e
fef7d230 _lwp_start (fec30400, 0, 0, fec2dff8, fef7d230, fec30400)


-----------------lwp# 5 / thread# 5 -------------------fef7d2a9
lwp_park (0, 0, 0)
fef733a0 sema_wait (8060d5c) + d
fefb565b sem_wait (8060d5c) + 27
08050a36 createAB (0) + 26
fef7cf3f _thr_setup (fec30800) + 4e
fef7d230 _lwp_start (fec30800, 0, 0, feb2dff8, fef7d230, fec30800)


-bash-3.00$
```

Пример 4.2. Выдача программы pstack(1)

Однако наиболее мощным средством анализа поведения программ считаются отладчики.

Примечание

Для использования отладчиков программу рекомендуется компилировать и собирать с ключом `-g` (генерировать отладочную информацию). Это позволяет вести отладку в терминах строк исходного кода и переменных. Без отладочной информации доступна только

отладка деассемблированного кода. Это в равной мере относится ко всем отладчикам, рассматриваемым далее на этой лекции.

Наиболее распространенный отладчик для систем семейства Unix, gdb (GNU Debugger) поддерживает многопоточную отладку на большинстве современных ОС, однако при сборке отладчика эту возможность можно выключить.

Проверить поддержку многопоточности в вашей версии gdb можно при помощи команды `info threads` (см. пример 1.1 в лекции 1).

Отладка многопоточных программ в gdb

Gdb назначает всем нитям отлаживаемой программы собственные числовые идентификаторы, обычно совпадающие с порядком создания этих нитей, но не обязательно с системными идентификаторами соответствующих нитей и LWP. Эти идентификаторы выводятся во второй колонке вывода команды `info threads`. Все остальные команды, оперирующие нитями, используют эти идентификаторы для ссылки на конкретные нити.

Одна из нитей программы является текущей с точки зрения gdb. Эта нить отмечается символом `*` в выводе команды `info threads`. Переключить текущую нить можно командой `thread threadno`, где `threadno` – идентификатор нити в терминах gdb. Кроме того, если нить, отличная от текущей, достигла точки останова (`breakpoint`), отладчик автоматически переключается на нее.

Команды пошагового исполнения(`step`, `next`, `finish` и др.) исполняются в контексте текущей нити. Также, команды выдачи значений локальных переменных и стека оперируют стеком текущей нити.

При остановках программы по любой причине (точка останова, пошаговое исполнение и др.) отладчик останавливает исполнение всех нитей программы. При продолжении исполнения продолжается исполнение всех активных нитей программы. Однако способа синхронизовать это исполнение, например пошагово исполняя по одной строке исходного кода в нескольких нитях, не существует. На

практике, при пошаговом исполнении одной из нитей остальные нити будут останавливаться не на границах операторов исходного кода, а на произвольных командах.

Точки останова (`breakpoint`) по умолчанию создаются для всего процесса, т.е. приводят к остановке любой нити, достигшей этой точки. Однако можно создавать точки останова для отдельных нитей. Точный синтаксис команды установки точки останова таков: `break linespec [thread threadno] [if condition]`. Квадратными скобками отмечены необязательные параметры команды.

`linespec` определяет номер строки исходного кода или адрес команды, на которой будет установлена точка. Gdb поддерживает несколько способов задания номера строки (номер строки в файле, номер строки в функции, номер строки относительно текущей). `threadno` представляет собой идентификатор нити, назначенный gdb. `condition` представляет собой условие, заданное при помощи подмножества языка программирования, при котором будет срабатывать точка останова. Точки останова без условий срабатывают при любой передаче управления на точку останова.

При отладке программ на языках C/C++ в качестве языка, на котором задаются условия, используется подмножество C. В действительности, условия можно задавать на любом из языков, поддерживаемых gdb; gdb определяет язык, на котором написан модуль программы, по расширению файла исходного текста. Кроме того, язык можно менять явно командой `set language`. Команда `set language` без параметров выводит список всех языков, поддерживаемых вашей версией gdb.

Полная документация по gdb может быть найдена на домашней странице проекта GDB по адресу ссылка: http://sourceware.org/gdb/documentation/ - http://sourceware.org/gdb/documentation/.

Отладка многопоточных программ в dbx

Dbx – это отладчик, входящий в состав SunStudio. Он может использоваться как самостоятельный отладчик с командным

интерфейсом либо как часть графического экранного отладчика SunStudio. Полная документация по командному режиму dbx доступна по адресу ссылка: http://docs.sun.com/app/docs/doc/819-3683 - http://docs.sun.com/app/docs/doc/819-3683.

Dbx в командном режиме имеет возможность эмулировать `gdb` ; для этого следует исполнить команду `gdb on`. После этого отладчик начнет воспринимать значительное подмножество команд gdb, но перестанет понимать команды `dbx`. Вернуться в основной режим можно командой `gdb off`.

В "родном" режиме `dbx` список нитей текущего процесса выводится командой `threads` (см. пример 4.3)

```
(dbx) threads

t@1 a l@1 ?() running in main()

t@2 ?() asleep on 0xef751450 in_swtch()

t@3 b l@2 ?() running in sigwait()

t@4 consumer() asleep on 0x22bb0 in _lwp_sema_wait()

*>t@5 b l@4 consumer() breakpoint in Queue_dequeue()
t@6 b l@5 producer() running in _thread_start()
(dbx)
```

Пример 4.3. выдача команды threads

Символы `*` или `o` отмечают нить, в которой произошло событие, требующее внимания отладчика. В примере 4.3 это точка останова. Символ `>` отмечает текущую (с точки зрения отладчика) нить. В отличие от `gdb`, эти нити могут не совпадать. `t@number` обозначает

идентификатор нити в терминах dbx. `l@number` обозначает LWP, к которому привязана или в рамках которого исполняется нить.

Внимательный читатель может догадаться, что в примере 3 приводится результат отладки программы, использующей старую реализацию POSIX Threads API, в которой нитей могло быть больше, чем LWP.

За `l@number` выводится стартовая функция нити (функция, которая была передана в качестве параметра `pthread_create(3C)`), текущее состояние нити и имя функции, которую нить исполняет в данный момент.

Список LWP процесса может быть выведен командой *`lwps`* (см. пример 4.4).

```
dbx) lwps
l@1 running in main()
l@2 running in sigwait()
l@3 running in _lwp_sema_wait()


 *>l@4 breakpoint in Queue_dequeue()
l@5 running in _thread_start()
(dbx)
```

Пример 4.4. Вывод команды lwps

Как и gdb, dbx является синхронным отладчиком. Это означает, что при остановке любой из нитей процесса по любой причине останавливаются все нити этого процесса. Поэтому состояние нити `running` означает, что эта нить готова к исполнению, но остановлена отладчиком.

Переключение текущей нити осуществляется командой `thread threadid`. Команда `thread` без параметров выводит номер текущей нити. При помощи команды `thread` можно осуществлять ряд других операций над нитями. Полный список опций команды `thread` приводится в документации или во встроенной справке dbx, которую можно вызвать командой `help thread`.

Команды пошагового исполнениями анализа содержимого стека выполняются по отношению к текущей нити. При пошаговом исполнении на время каждого шага возобновляется исполнение всех готовых к исполнению нитей.

Отладка многопоточных программ в SunStudio

Интегрированная среда разработки SunStudio 11 основана на NetBeans и содержит многооконный экранный отладчик. В действительности этот отладчик представляет собой графический интерфейс к отладчику dbx. В одном из окон доступна командная строка dbx и можно исполнять произвольные команды dbx.

На рис 4.1 приводится снимок экрана SunStudio при отладке многопоточной программы.

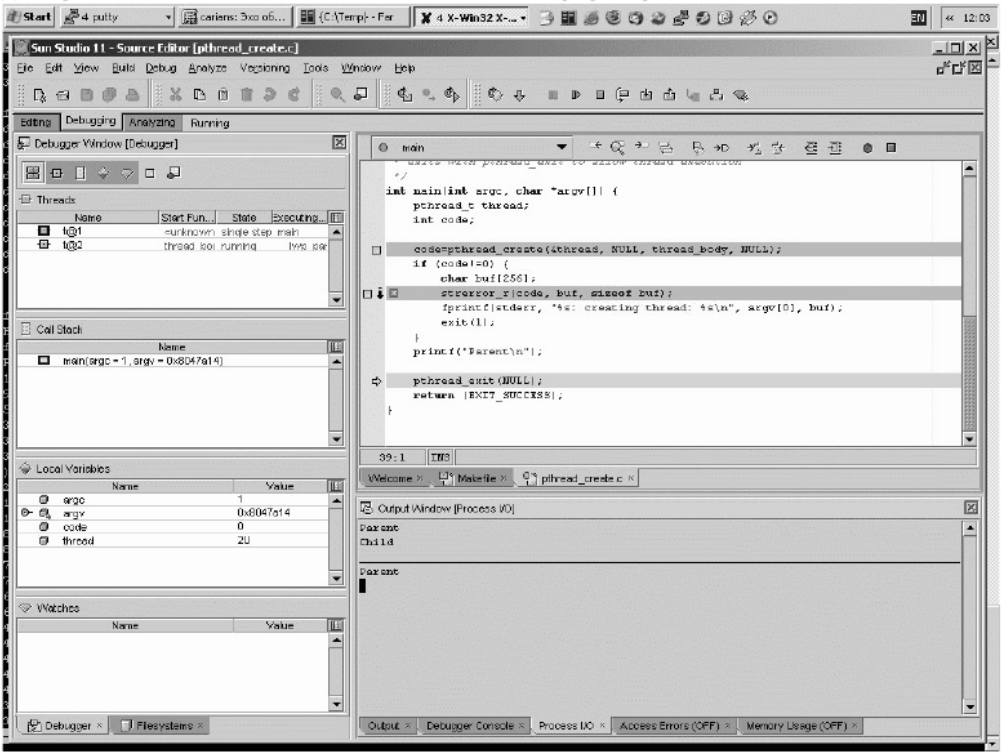


Рис. 4.1. Экран SunStudio 11 при отладке многопоточной программы

В верхнем окне левой панели (окно так и озаглавлено – Threads) виден список всех нитей процесса с указанием их состояний и текущей исполняемой функции. Формат, в котором выводится описание каждой из нитей, аналогичен формату вывода команды threads отладчика dbx. Текущая нить выделена.

На рис. 4.2 приводится вид той же отладочной сессии после переключения текущей нити.

Характерная бледно-розовая отметка строки `printf("Child\n");` обозначает, что исполнение нити остановлено в этой строке, но не точно на границе строки. Стек вызовов, выведенный в окне "Call Stack" позволяет увидеть, где именно остановлено исполнение.

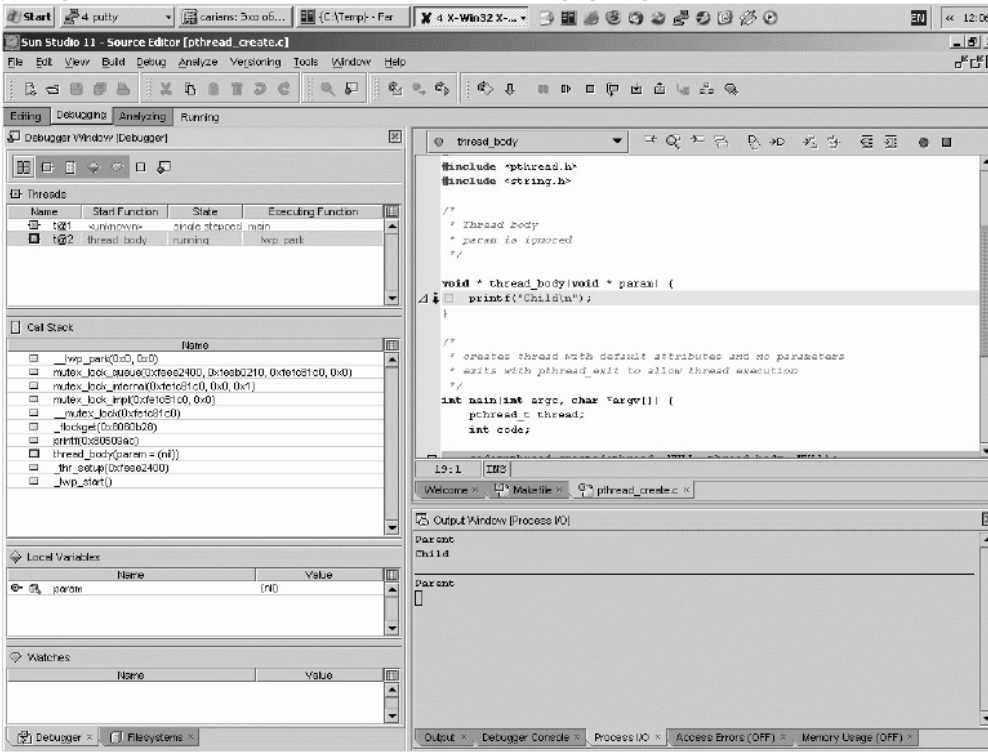


Рис. 4.2. Экран SunStudio 11 после переключения текущей нити

Нити и стандартные библиотеки Unix

По завершении этого раздела вы сможете: использовать стандартные библиотеки или их аналоги в многопоточных программах, находить в документации информацию о том, является ли данная функция или группа функций, использовать сигналы и fork в многопоточных программах.

Копоть-сажу смыл под душем,

Съел холодного язя

И инструкции послушал -

Что там можно, что нельзя.

Там у них бывает лучше бытово,

Так чтоб я не отчубучил не того -

Он мне дал прочесть брошюру как наказ,

Чтоб не вздумал жить там сдуру, как у нас.

В. Высоцкий

Что означает thread-safe?

Термин *thread-safe* не имеет общепринятого перевода на русский язык. Дословно это словосочетание следует переводить как "безопасный для использования в многопоточной программе". Чаще всего этот термин применяется к библиотечным функциям и библиотекам в целом.

Проблема состоит в том, что многие библиотеки используют те или иные внутренние переменные. Некоторые библиотеки - например `malloc(3C)/free(3C)` - используют довольно сложные внутренние структуры данных. Для нормальной работы библиотечных функций эти структуры данных должны удовлетворять определенным

требованиям, то есть быть согласованными (consistent) или (что то же самое) целостными. При этом во время работы функции могут временно нарушать согласованность своих внутренних структур данных. При нормальной работе к моменту завершения функции согласованность восстанавливается.

Но если во время работы функции в одной нити другая нить вызовет функцию этой же библиотеки - не обязательно ту же самую, но полагающуюся на целостность той же внутренней структуры данных - скорее всего работа функции приведет к ошибкам. Для примера рассмотрим ситуацию параллельного вывода в стандартный поток вывода. Функция printf(3S) формирует текстовые данные в соответствии с заданным форматным спецификатором и помещает их в буфер вывода. При использовании длинных форматных спецификаторов и большого числа параметров printf(3S) работает долго. Если во время его работы другая нить тоже позовет printf(3S), то в лучшем случае данные, формируемые каждой из нитей, будут перемешаны в выходном буфере. В худшем случае вывод одной из нитей будет записан в буфер поверх вывода другой нити.

Для решения этой проблемы *thread-safe* реализация printf(3S) должна каким-то образом защищать себя от многократного (реентрантного) вызова. Для этого используются мутексы или блокировки чтения-записи, которые мы будем рассматривать на следующих лекциях.

Еще более сложная проблема возникает, если функция должна сохранять некоторые данные между своими вызовами. В качестве примера рассмотрим функцию strtok(3C). Функция strtok(3C) при первом вызове разбивает строку-параметр на токены в соответствии с заданными разделителями и возвращает первый токен. При последующих вызовах она возвращает второй, третий и т.д. токены. Для этого она должна где-то хранить указатель на текущий токен; однопоточные версии стандартной библиотеки языка C обычно используют для этого статическую переменную. Если в другой нити эта функция будет вызвана с другой начальной строкой, этот указатель будет перезаписан. Результат следующего вызова strtok(3C) в первой нити при этом будет сильно отличаться от того, что, скорее всего, ожидал программист, разрабатывавший программу этой нити. Чтобы

защититься от возникающих при этом неприятностей, недостаточно защитить `strtok(3C)` от реентрантного вызова, нужно сделать что-то гораздо более сложное.

Один из вариантов (реально использованный в библиотеке Solaris 10) обсуждался на предыдущей лекции и состоит в использовании `ThreadSpecificData`. Другой вариант состоит в изменении API. Для `strtok(3C)` предлагается *`thread-safe`* версия `strtok_r(3C)` ; эта функция хранит указатель на следующий токен не во внутренней переменной, а в ячейке памяти, которую должна предоставить вызывающая функция. Уже на этих примерах видно, что задача обеспечения безопасности с точки зрения многопоточности - сложная задача, не имеющая универсального решения. Видно также, что проблемы с многопоточностью часто заложены уже на уровне описания семантики функций.

При разработке стандартной библиотеки языка C и многих других стандартных или просто широко распространенных библиотек традиционных Unix-систем требования многопоточности не принимались во внимание. В Solaris 10 большинство из этих библиотек переработаны для соответствия требованиям многопоточности, но такая переработка не завершена и не для всех функций выполнена в совершенстве. Чтобы программист мог знать, какие из функций можно использовать в многопоточной программе, какие - нельзя, а какие можно, но с ограничениями, в страницах системного руководства Solaris (man(1)) приводится информация о многопоточности. Эта информация приводится в секции АТРИБУТЫ (`ATTRIBUTES`) (см. пример 5.1).

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Unsafe

Пример 5.1. Секция ATTRIBUTES страницы руководства getch(3CURSES)

Описание возможных атрибутов приводится в attributes(5). С точки зрения многопоточности, наиболее интересный атрибут - это MT-Level (уровень многопоточности). Допустимые уровни многопоточности перечислены далее. Функции, для которых MT-Level не указан, считаются Safe.

Unsafe - функция или набор функций используют незащищенные глобальные или статические данные. Вызывающая программа обязана теми или иными средствами гарантировать, что никакие функции из этого набора не будут одновременно вызваны из различных потоков. Если такая гарантия не будет выполнена, результаты непредсказуемы. Некоторые из *Unsafe* функций имеют реентерабельные аналоги. Обычно имена функций аналогов получаются добавлением суффикса _r (от *reentrant* или reenterable) к имени функции.

Safe - функция или набор функций могут вызываться из нескольких потоков. Однако при этом возможны определенные ограничения. Так, открытие и закрытие файла через системные вызовы open(2) и close(2) воздействует на все нити процесса.

Закрытие файла, с которым работают другие нити, может приводить к нежелательным последствиям для этих нитей, хотя сам по себе системный вызов close(2) является *thread-safe*.

MT-Safe - функция или набор функций полностью подготовлены для работы в многопоточной среде. Библиотека защищает свои локальные данные (если они есть) при помощи средств взаимоисключения и обеспечивает разумный уровень параллелизма, т.е. не удерживает средства взаимоисключения дольше, чем это необходимо.

MT-Safe with Exceptions (безопасно с исключениями) - функция может быть небезопасна при некоторых вариантах использования. Информация о том, при каких именно сценариях использования функция может быть небезопасна, содержится в секциях руководства NOTES и/или USAGE.

Asynch-Signal-Safe - функция может вызываться в многопоточной программе из обработчиков сигналов. Проблема в том, что если функция обеспечивает уровень MT-Safe за счет использования

примитивов взаимоисключения, то реентрантный вызов этой функции из обработчика сигнала может привести к мертвой блокировке. Существует ряд способов решенияи обхода этой проблемы, но они довольно сложны и не всегда применимы, поэтому большинство `MT-Safe` функций не являются Asynch-Signal-Safe.

`Fork-Safe` - функция безопасна, даже если во время работы этой функции другая нить процесса вызовет `fork(2)`. Проблемы, которые могут при этом возникать, и способы их решения обсуждаются далее на этой лекции.

`Deferred-Cancel-safe` - функция безопасна для использования в нитях, работающих в режиме отложенного прерывания (`PTHREAD_CANCEL_DEFERRED`). `Asynchronous-Cancel-Safe` - функция безопасна для использования в нитях, работающих в режиме асинхронного прерывания (`PTHREAD_CANCEL_ASYNCHRONOUS`). Подразумевает `Deferred-Cancel-Safe`.

Примеры thread-safe интерфейсов

Рассмотрим несколько примеров безопасных интерфейсов.

Readdir(3C) и readdir_r(3C)

Функция `readdir(3C)` возвращает очередную запись каталога файловой системы. Эта имеет проблему с многопоточностью, заложенную на уровне описания интерфейса. Она возвращает указатель на `struct dirent`. Чтобы упростить для вызывающей программы управление памятью, эта функция всегда возвращает указатель на одну и ту же структуру (внутренний буфер). Последующий вызов `readdir(3C)` перезапишет значение, возвращенное предыдущим вызовом. Поскольку `readdir(3C)` обычно используется для последовательного сканирования каталога, это поведение в большинстве случаев удовлетворительно. Но оно совершенно неприемлемо для многопоточных программ.

Для обхода этой проблемы была предложена реентерабельная версия

`readdir(3C)` - `readdir_r(3C)`. Эта функция получает указатель на буфер, в котором следует разместить описание записи каталога.

Интерфейс `readdir_r(3C)` на первый взгляд кажется простым и логичным, однако при его использовании важно знать о существовании одной проблемы. Проблема эта состоит в том, что `struct dirent` - структура переменного размера. Буфер, размер которого равен `sizeof(struct dirent)`, недостаточен для размещения структуры данных, которая будет возвращена вызовом `readdir_r(3C)`.

Страница руководства `readdir_r(3C)` рекомендует вычислять размер буфера по формуле `sizeof(struct dirent)+pathconf(directory, _PC_NAME_MAX)`. Системный вызов `pathconf(2)` с параметром `_PC_NAME_MAX` возвращает размер максимального допустимого имени файла в файловой системе, в которой размещен каталог `directory`. Получение этой информации при помощи системного вызова дает определенные преимущества; в частности, это позволяет избежать проблем, которые могут возникнуть в будущих версиях операционной системы, которые могут поддерживать файловые системы с большой длиной имени файла.

Однако данный способ сопряжен и с определенными недостатками. Наиболее очевидный недостаток состоит в том, что вычисление размера структуры во время исполнения лишает программиста возможности размещать структуру в статическом буфере и вынуждает использовать `malloc(3C)`, `alloca(3C)` или динамические массивы `C99`.

Менее очевидный, но более серьезный недостаток, состоит в том, что `readdir_r(3C)` не знает длины вашего буфера. Таким образом, если вы неверно вычислите длину буфера, это может привести к срыву буфера. Наиболее вероятный сценарий такой ошибки происходит, если программист предполагает, что все подкаталоги каталога `directory` имеют `NAME_MAX`, равную `pathconf(directory, _PC_NAME_MAX)`. Однако если один из подкаталогов является символической ссылкой или точкой монтирования, это может быть неверно.

Произвольный доступ к файлу

Рассмотрим еще один источник проблем с многопоточностью, присутствующий в традиционном API Unix-систем. Системные вызовы `read(2)` и `write(2)` при работе с регулярными файлами и некоторыми устройствами используют понятие текущей позиции. Эти системные вызовы начинают чтение и запись с текущей позиции и переставляют текущую позицию на конец прочитанного или записанногоучастка файла.

Кроме того, текущую позицию можно переставлять системным вызовом `lseek(2)`.

Очевидно, что последовательность `lseek(..) ; write(..) ;` не может быть *thread-safe*, да и с безопасностью самих вызовов `read(2)` и `write(2)` не все просто.

Для решения этой проблемы POSIX Thread API предоставляет вызовы `pread(2)` и `pwrite(2)`.

В отличие от традиционных вызовов `read(2)` и `write(2)`, эти вызовы имеют четвертый параметр типа `off_t`. Вызов `pread(file, buffer, size, offset)` ; можно рассматривать как атомарно исполняющуюся последовательность вызовов `lseek(file, offset, SEEK_SET); read(file, buffer, size);`.

При использовании `pread(2)` и `pwrite(2)` с устройствами и псевдоустройствами (например, трубами или сокетами), которые не поддерживают `lseek`, четвертый параметр игнорируется.

Сигналы в многопоточном процессе

Сигналы в системах семейства Unix могут быть поделены на две большие категории: синхронные и асинхронные. Синхронные сигналы возникают при исполнении программой определенных операций и, таким образом, привязаны к определенной точке кода и определенному потоку. Асинхронные сигналы возникают при событиях, внешних по

отношению к процессу, и не имеют такой привязки.

Примеры синхронных сигналов - `SIGFPE` (FloatingPointException,ошибка операции с плавающей точкой или целочисленное деление на ноль), `SIGSEGV` (Segmentation Violation, ошибка доступа к защищенной странице или сегменту памяти), `SIGBUS` (*`Bus [Error]`*, ошибка шины - в процессорах `SPARC` возникает при обращении к невыровненным словам), `SIGPIPE` (запись в трубу или сокет, другой конец которых закрыт).

Примеры асинхронных сигналов - `SIGINT` (генерируется терминальным драйвером при получении символа Ctrl-C), `SIGALARM` (генерируется таймером астрономического времени), `SIGTERM` (генерируется по умолчанию шелловской командой `kill(1)`). Важно понимать, что если синхронный сигнал будет сгенерирован системным вызовом `kill(2)` или шелловской командой `kill(1)`, он будет обрабатываться как асинхронный. Обработчики синхронных сигналов вызываются в той нити, в которой этот сигнал возник. Обработчики асинхронных сигналов вызываются в любой нити, способной обработать этот сигнал.

Системные вызовы `signal(2)`, `sigset(2)` и `sigaction(2)` устанавливают глобальный обработчик сигнала, исполняющийся во всех нитях процесса. Штатного способа установить собственный обработчик сигнала в пределах нити не существует. Однако вопрос о том, в какой именно нити будет вызван обработчик, важен с нескольких точек зрения. Во первых, если обработчик сигнала использует `longjmp(3C)`, то он должен вызываться в той нити, в которой был исполнен соответствующий `setjmp(3C)`. В противном случае, `longjmp(3C)` приведет к разрушению стека нити и аварийному завершению программы. Аналогичная проблема может возникать, если обработчик сигнала генерирует исключения C++, только в этом случае проблема, скорее всего, ограничится необработанным исключением.

Во вторых, если нить и обработчик сигнала используют функции, не являющиеся `AsynchSignal - Safe`, им необходимо как-то координировать исполнение этих функций, скорее всего (но не обязательно) гарантируя, что они никогда не будут вызваны в одной нити. Для решения обеих проблем необходимо управлять тем, в каких

нитях будут вызваны обработчики сигналов. Для этой цели существуют два средства - маска сигналов нити и функция `pthread_kill(3C)`.

Маска сигналов нити функционально аналогична маскесигналов процесса, с той очевидной разницей, что она работает в пределах одной нити. Маска представляет собой множество (набор) сигналов. В большинстве Unix-систем это множество реализовано в виде битовой маски, но API для работы с масками, определяемое стандартом POSIX, допускает и другие реализации. Маска представляет собой непрозрачный тип `sigset_t`, над которым определены некоторые теоретико-множественные операции, описанные на странице руководства `sigsetops(3C)`. Если сигнал установлен в маске (маскирован), то нить не будет обрабатывать этот сигнал. Если в масках одной или несколькихдругих нитей этого сигнала нет, обработчик будет вызван в одной из этих нитей. Если сигнал маскирован во всех нитях, он будет задержан до момента, пока в одной из нитей он не будет размаскирован.

Маскирование сигналов на уровне процесса (при помощи маски сигналов процесса) приведет к тому же результату.

Маска сигналов нити наследуется у родителя. Операции над маской сигналов нити осуществляются библиотечной функцией `pthread_sigmask(3C)`. Эта функция аналогична системному вызову `sigprocmask(2)` и имеет похожие три параметра:

- `int how` - определяет операцию, которую необходимо выполнить над маской. Может принимать три значения: `SIG_BLOCK` (сигналы, перечисленные в параметре `set`, будут заблокированы, остальные - оставлены без изменения), `SIG_UNBLOCK` (сигналы, перечисленные в параметре `set`, будут разблокированы, остальные - оставлены без изменения) и `SIG_SETMASK` (маска сигналов нити будет заменена на значение параметра `set`).
- `sigset_t * set` - входной параметр, набор сигналов. Точное значение этого параметра определяется значением параметра `how`. Если в качестве этого параметра передать нулевой указатель, параметр `how` будет проигнорирован и маска изменена не будет.

  Это можно использовать для получения текущего значения маски

без ее изменения.

- `sigset_t *oset` - выходной параметр, в нем сохраняется старое значение маски сигналов нити. Если в качестве этого параметра передать нулевой указатель, старое значение маски будет потеряно навсегда. Используя маски сигналов нитей можно создать выделенную нить или группу нитей, которые будут обрабатывать все сигналы.

Если необходимо обработать сигнал в определенной нити, можно использовать функцию `pthread_kill(3C)`. Эта функция посылает указанныйсигнал указанной нити. Однако если этот сигнал в этой нити замаскирован, он не будет доставлен.

Альтернативой традиционным обработчикам сигналов является системный вызов `sigwait(2)`.

Этот вызов может обрабатывать один или несколько сигналов, определяемых параметром входным параметром `sigset_t *set`. Если нить не имеет ни одного ожидающего обработки сигнала из множества, заданного параметром `set`, `sigwait(2)` блокируется. Если такой сигнал появляется (даже если он маскирован), `sigwait(2)` извлекает этот сигнал из набора или очереди сигналов, ожидающих обработки, и возвращает управление. Таким образом, `sigwait(2)` представляет собой своего рода синхронный обработчик для одного или нескольких сигналов. В однопоточной программе такой обработчик был бы неудобен, но в многопоточной программе использование `sigwait(2)` позволяет упростить архитектуру межпоточного взаимодействия и решить ряд проблем, в том числе (но не только) связанных с функциями, не являющимися *`Asynch-Thread-Safe`*.

Обработчик сигнала `sigwait(2)` имеет приоритет перед традиционными функциямиобработчиками, устанавливаемыми при помощи `signal(2)/sigset(2)/sigaction(2)`. На практике означает, что если в программе есть обработчик сигнала, установленный `signal(2)`, и при этом одна из нитей заблокирована в `sigwait(2)`,то сигнал будетобработан при помощи `sigwait(2)`. Однако не следует злоупотреблять этим фактом - ведь нить не все время

будет заблокирована в `sigwait(2)`, и в эти моменты времени может быть вызван традиционный обработчик.

В старых версиях Solaris, `sigwait(2)` возвращал номер сигнала вкачестве кода возврата; в Solaris 10, если программа компилируется с ключом - `D_POSIX_THREAD_SEMANTIC`, `sigwait(2)` возвращает номер сигнала во втором (выходном) параметре. Код возврата при этом соответствует коду ошибки или равен `0`, если ошибки не было.

Если несколько нитей исполняют `sigwait(2)` с пересекающимися множествами сигналов и поступает сигнал, принадлежащий к пересечению этих множеств, то только одна нить получит этот сигнал, а остальные нити останутся заблокированными.

fork(2) в многопоточной программе

Системный вызов `fork(2)` в Unix-системах - это основноесредство создания процессов. Как известно, этот системный вызов создает копию родительского процесса. В начальный момент образ созданного процесса отличается от родительского только кодом возврата `fork(2)`.

Из такого описания следует, что `fork(2)` в многопоточной программе должен был бы приводить к копированию всех нитей родительского процесса. Собственно, в старых версиях Unix SVR4, в том числе и в старых версиях Solaris, так оно и было. Понятно, что такое поведение не очень логично при большинстве традиционных применений `fork(2)`, поэтому в стандарте POSIX Thread API такое поведение не предусмотрено и требуется, чтобы при `fork(2)` дублировалась только нить, в которой произошел вызов `fork(2)`.

Для обеспечения совместимости с приложениями, рассчитанными на старую семантику, Solaris 10 предоставляет системный вызов `forkall(2)`. Этот вызов не соответствует стандартам и не имеет аналогов в других реализациях POSIX Threads API.

Однако семантика `fork(2)`, при которой дублируется только одна нить, тоже может приводить к проблемам. Действительно, другие нити в момент `fork` чем-то занимались и могли удерживать какие-то

примитивы взаимоисключения. Если после `fork` дублированной нити потребуется какой-то из этих примитивов, это приведет к мертвой блокировке. Автоматически освобождать все примитивы взаимоисключения в момент `fork` тоже некорректно, ведь этоприведет к доступу к несогласованным структурам данных. Поэтому все примитивы взаимоисключения и синхронизации в дочернем процессе сохраняются в том же состоянии, в каком они были в родительском процессе на момент `fork`.

Для решения этой проблемы POSIX Thread API предоставляет функцию `pthread_atfork(3C)`. Эта функция имеет три параметра типа "указатель на функцию", обозначаемые как `prepare`, `parent` и `child`.

Функция `prepare` вызывается после вызова `fork(2)`, но перед собственно созданием дочернего процесса. Функции `parent` и `child` вызываются, соответственно, в родительском и дочернем процессе.

Многократный вызов `pthread_atfork(3C)` приводит к регистрации нескольких обработчиков, при этом обработчики `prepare` вызываются в порядке LIFO (Last In First Out, то есть в порядке, обратном тому, в котором делались вызовы `pthread_atfork(3C)`), а `parent` и `child` - в порядке FIFO (First In First Out, то есть в том же порядке, что и `pthread_atfork(3C)`).

Стандартное решение проблемы наследования блокировок, рекомендуемое на странице системного руководства `pthread_atfork(3C)`, состоит в следующем. Если вы реализуете какую-либо библиотеку, использующую примитивы взаимоисключения или синхронизации, и хотите сделать ее `Fork-Safe`, вы должны зарегистрировать обработчики `atfork`. Вы можете реализовать отдельную тройку обработчиков на каждый примитив взаимоисключения или одну тройку на всю библиотеку. По ряду очевидных причин предпочтителен второй вариант, поэтому далее мы будем рассматривать его.

Скорее всего, ваша библиотека должна использовать определенный порядок захвата примитивов взаимоисключения для избежания мертвой блокировки. Ваш обработчик `prepare` должен захватить все

эти примитивы взаимоисключения в этом порядке. Если все структуры данных вашей библиотеки защищены соответствующими примитивами, то ничего более делать не требуется. Обработчики `parent` и `child` должны освободить все эти примитивы синхронизации. При таком подходе нередко удается реализовать эти два обработчика при помощи одной функции.

Выполнение этих действий гарантирует, что к моменту создания дочернего процесса все ресурсы вашей библиотеки находятся в согласованном состоянии, а к моменту возврата из `fork(2)` все блокировки вашей библиотеки сняты.

Примечание

Если ваша библиотека удерживает какие-то из своих блокировок в промежутке между вызовами своих функций, описанный выше подход может привести к мертвой блокировке. Действительно, если пользователь заставит вашу библиотеку захватить блокировку, а затем вызовет `fork`, то ваш обработчик `prepare` попытается повторно захватить ту же самую блокировку. Это и есть простейшая форма мертвой блокировки.

Этот сценарий - одна из причин (неединственная и дажене основная),по которой удерживать блокировки в промежутках между вызовами библиотечных функций - плохая практика.

Если обстоятельства все-таки вынуждают вас прибегать к такой практике, вам следует разработать более сложную схему обеспечения `Fork-Safety`.

Мутексы

В ходе этой лекции вы изучите: задачи, для решения которых необходимы мутексы, создание и уничтожение мутексов, атрибуты мутексов, использование мутексов в качестве примитива взаимоисключения, критические секции и взаимоисключение.

Пытлив умом был Федор, но не мудр

Не разгадав явление природы

Решил он, что ларек снесли за ночь

В. Шинкарев

В нескольких предыдущих лекциях мы видели, что основная проблема при разработке многопоточных приложений - это обеспечение согласованности разделяемых структур данных. На практике эта проблема решается главным образом при помощи концепций критических секций и взаимоисключения.

Критическая секция -это участок кода программы, вовремя исполнения которого программа либо полагается на целостность разделяемой структуры данных, либо тем или иным образом нарушает эту целостность. Для примера рассмотрим операции над отсортированным массивом без дублирующихся элементов. Основные операции над таким массивом - поиск, вставка и исключение элемента. Поиск в отсортированном массиве удобно осуществлять дихотомическим или, что то же самое, двоичным алгоритмом. Этот алгоритм работает очень быстро, но если условие сортировки по какой-то причине будет нарушено, попытка дихотомического поиска завершится неудачно.

Все способы вставки и исключения элемента в отсортированном массиве предполагают либо [временное] нарушение условия сортировки, либо появление в массиве дубликатов, либо то и другое одновременно. Поэтому все перечисленные операции являются критическими секциями. Для обеспечения нормальной работы этих операций нам необходимо тем или иным способом гарантировать, что они не будут выполняться одновременно над одним и тем же массивом.

Это и называется взаимоисключением.

Примечание

Внимательный читатель может отметить, что в отсортированном массиве очень сложно реализовать параллельное исполнение нескольких вставок или удалений, и практически невозможно реализовать параллельное исполнение вставки и поиска, но несколько поисков вполне могут выполняться одновременно.

Разделяемые данные являются основным средством взаимодействия между нитями. Собственно, если бы нам были не нужны разделяемые данные, мы могли бы использовать процессы вместо нитей.

Главная сложность при определении критической секции состоит в том, что это, строго говоря, не формализованное понятие. Для каждого конкретного алгоритма можно точно определить, какие структуры данных с его точки зрения являются согласованными. Но критические секции связаны с данными, а с точки зрения данных невозможно определить, какие алгоритмы будут использоваться для работы с этими данными. Поэтому невозможно автоматическиопределять критерии согласованности данных и границы критических секций в коде. Все известные методики работы с разделяемыми данными - как с данными, размещенными в оперативной памяти, так и с базами данных - в конечном итоге опираются на то, что программист должен определить и явно указать в коде границы критических секций. Например, в реляционных СУБД критические секции соответствуют транзакциям. Использовать простые флаговые переменные для защиты критических секций невозможно. Действительно, последовательность команд, включающая в себя проверку флаговой переменной и ее установку, сама представляет собой критическую секцию и нуждается в защите.

Существует ряд решений задачи взаимоисключения на нескольких флаговых переменных, например алгоритм Деккера, который состоит в использовании двух флаговых переменных. Каждая из взаимодействующих нитей устанавливает "свой" флаг, а проверяет "чужой".

Этот алгоритм сложно распространить на взаимодействие произвольного количества нитей, поэтому на практике он очень редко

применяется.

Большинство современных процессоров в той или иной форме предоставляют атомарную (неделимую) операцию проверки и установки флага. Например, в процессорах x86 это команда `EXCHG` (exchange, обменять), которая обменивает значения регистра и ячейки памяти.

Единственный разумный способ ожидать освобождения флаговой переменной - это холостой цикл. Исполняющая такой цикл программа занимает процессор, но не делает ничего полезного. Примитив взаимоисключения, использующий проверку флага в холостом цикле и его проверку и установку при помощи специальной атомарной команды процессора, называется спинлоком (spinlock). Спинлоки используются многими операционными системами для взаимодействия между копиями ядра ОС, исполняющимися на разных процессорах многопроцессорной машины, а также в некоторых приложениях жесткого реального времени. POSIX Thread API также предоставляет спинлоки, но поддержка соответствующей группы типов и функций опциональна (не обязательна для всех реализаций). Solaris поддерживает спинлоки POSIX, но в нашем курсе мы их изучать не будем.

Для взаимодействия между пользовательскими процессами и нитями современные ОС и многопоточные среды программирования предоставляют более сложные примитивы, объединяющие проверку флага и засыпание, пробуждение и установку флага и, наконец, сброс флага и пробуждение другой нити в атомарные операции.

Под словом "примитив" мы в данном случае понимаем непрозрачный тип данных, над которым определен фиксированный набор операций.

Мутекс является одним из таких примитивов. Слово мутекс (mutex) происходит от сокращения словосочетания mutual exclusion - взаимное исключение. В некоторых русскоязычных публикациях эти объекты также называют мьютексами. Такая транскрипция ближе к правильному английскому произношению этого слова.

Мутекс может находиться в двух состояниях - свободном и захваченном. Над мутексом определены две основные операции - блокировка (lock,

захват, asquire) и снятие (unlock, освобождение, release). Блокировка свободного мутекса приводит к его переводу в захваченное состояние. Попытка блокировки захваченного мутекса приводит к засыпанию (блокировке) нити, которая пыталась выполнить эту операцию. Освобождение свободного мутекса - недопустимая операция; в зависимости от особенностей реализации эта операция может приводить к непредсказуемым последствиям или к ошибке или просто игнорироваться. Освобождение занятого мутекса приводит к переводу мутекса в свободное состояние; если в этот момент на мутексе были заблокированы одна или несколько нитей, одна из этих нитей пробуждается и захватывает мутекс.

Главной особенностью мутексов является тот факт, что захват и освобождение мутекса должны производиться одной нитью. Поэтому можно говорить о том, что нить, захватившая мутекс, владеет им.

Применение мутексов для решения задачи взаимоисключения очевидно. Мы должны связать с каждым разделяемым ресурсом мутекс. Когда нить входит в критическую секцию, связанную с ресурсом, она должна захватить мутекс, а когда выходит из нее - освободить.

Создание и уничтожение мутексов POSIX Thread Library

Мутексы в POSIX Thread API имеют тип `pthread_mutex_ t`. Это непрозрачный тип, операции над которым должны осуществляться соответствующими функциями библиотеки POSIX Threads. Внутренняя структура объектов этого типа не документирована и может различаться в разных реализациях и даже в разных версиях одной реализации POSIX Threads.

Перед использованием мутекс необходимо инициализировать. Это может делаться функцией `pthread_mutex_init(3C)` или присваиванием мутексу константы `PTHREAD_MUTEX_INITIALIZER`, определенной в `<pthread.h>`. Функция `pthread_mutex_init(3C)` получает два параметра, указатель на инициализируемый объект и указатель на описание атрибутов мутекса, структуру `pthread_mutex_attr_t`. Все параметры мутекса задаются в `pthread_mutex_attr_t`, которая

рассматривается далее на этой лекции. Результат выполнения остальных операций над неинициализированным мутексом не определен; это может приводить к блокировке нити на неопределенное время или к аварийному завершению процесса.

Перед освобождением памяти из-под мутекса мутекс необходимо уничтожить. Это делается функцией `pthread_mutex_destroy(3C)`. Операции над мутексом могут приводить к размещению дополнительной памяти или объектов ядра ОС, поэтому уничтожение мутекса без выполнения `pthread_mutex_destroy` может приводить к утечке памяти или исчерпанию системных ресурсов. Выполнение операции `pthread_mutex_destroy` над мутексом, на котором заблокирована одна или несколько нитей, приводит к неопределенным последствиям. Дальнейшие операции над этим мутексом также приводят к неопределенным последствиям.

Операции над мутексом

Над мутексом определены четыре основные операции: `pthread_mutex_lock(3C)`, `pthread_mutex_unlock(3C)`, `pthread_mutex_trylock(3C)` и `pthread_mutex_timedlock(3C)`. Семантика операций `lock` и `unlock` описывалась в начале лекции; единственное, что следует отметить - что `pthread_mutex_lock(3C)`, в отличие от большинства других блокирующихся операций, не является точкой прерывания.

`Pthread_mutex_trylock(3C)` пытается захватить мутекс; если он занят, операция возвращает ошибку `EAGAIN`.

`Pthread_mutex_timedlock(3C)` - блокировка с тайм-аутом. Эта функция пытается захватить мутекс и блокируется, если это невозможно, но не дольше чем до указанного момента. Если функция вернула управление по тайм-ауту, она возвращает ошибку `ETIMEOUT`. Solaris предоставляет также функцию `pthread_mutex_reltimedlock_np(3C)`, которая задает относительный тайм-аут, т.е. интервал времени от момента вызова.

Суффикс `_np` у имени функции обозначает, что эта функция не входит в стандарт POSIX.

По умолчанию операции над мутексами не осуществляют никаких проверок. При повторном захвате мутекса той же нитью произойдет мертвая блокировка. Результат других некорректных последовательностейопераций, например захвата мутекса в одной нити и освобождения в другой, или многократного освобождения, не определен стандартом, хотя в большинстве распространенных реализаций, в том числе и в Solaris 10, результаты таких операций обычно достаточно безобидны. Используя `pthread_attr_t`, при инициализации мутекса можно задать параметры, которые заставляют систему делать проверки при работе с мутексами и возвращать ошибки при некорректных последовательностях операций. Разумеется, операции над мутексами без проверок гораздо дешевле, чем с проверками.

Групповых операций над мутексами POSIX Thread Library не предоставляет. При практическом использовании мутексов следует иметь в виду, что мутекс похож на дорожный светофор: он указывает, что вам не следует пересекать дорогу, но не ставит вам физических препятствий. Иными словами, если программист по какой-то причине забыл захватить мутекс перед входом в критическую секцию или неверно определил границы критической секции, среда программирования сама по себе не сообщит ему об ошибке. Ошибки такого рода (так называемые ошибки соревнования, race condition) очень сложно обнаруживать при тестировании, особенно если проводить тестирование на машине с небольшим количеством процессоров. Даже если такая ошибка будет обнаружена, разработчику может оказаться нелегко воспроизвести ее.

Этот недостаток характерен для всех примитивов межпоточного взаимоисключения и синхронизации. Фактически, он является оборотной стороной того, что потоки разделяют память и никак не защищены друг от друга.

В поставку SunStudio 11 входит инструмент, который может использоваться для обнаружения ошибок соревнования в коде. Этот инструмент предполагает компиляцию и сборку программы со

специальными ключами. Компилятор при этом вставляет в код специальные инструкции для сбора статистики обращений к данным. После прогона программы можно просмотреть эту статистику и, в том числе, обнаружить операции над разделяемыми данными, не защищенные мутексом.

Мертвые блокировки

При использовании мутексов и большинства других примитивов взаимоисключения возникает проблема мертвых блокировок (`deadlock`). Иногда мертвые блокировки также называют тупиками (`dead end`). Простейшая форма мертвой блокировки - это ситуация, когда одна и та же нить два раза пытается захватить один и тотже мутекс.

В реальном коде такая мертвая блокировка может возникать, например, в следующей ситуации: программист пытается реализовать класс, все методы которого являются Thread- safe. Для этого он включает в структуру данных мутекс и захватывает и освобождает этот мутекс при каждом вызове метода своего объекта.

Затем у программиста возникает потребность ввызове собственного метода (см. пример 1.1):

```
class monitor {
private:
pthread_mutex_t mx;


public:
int method1()
{ pthread_mutex_lock(&mx); … pthread_mutex_unlock(&mx); }
int method2()
{


pthread_mutex_lock(&mx);
…
method1();
```

```
...
pthread_mutex_unlock(&mx);


}
}
```

Пример 1.1. Мертвая блокировка с одним ресурсом и одной нитью

POSIXThreadLibrary реализует несколько способов борьбы с этой ошибкой, которые рассматриваются далее на этой лекции.

Более сложный - и более опасный - сценарий возникновения такой же ошибки - вызов в обработчике сигнала функций, которые содержат внутренние блокировки и не являются Asynch-Signal-Safe.

Еще более сложный сценарий мертвой блокировки включает в себя минимум две нити и минимум два ресурса, например `A` и `B`. Если одна нить захватывает эти ресурсы в порядке `lock(A)` ; `lock(B)`, а другая - в порядке `lock(B)` ; `lock(A)` ;, при неудачном стечении обстоятельств нити могут заблокировать друг друга. Такая блокировка может включать в себя и более чем две нити; формальным критерием такой блокировки является возникновение цикла в графе ожидающих друг друга нитей.

При использовании мутексов, теоретически можно обнаруживать ошибки такого рода на уровне библиотеки, выстраивая граф ожидающих друг друга нитей и находя в нем циклы. На практике, стандарт POSIX не требует выполнения такой проверки даже для мутексов типа `PTHREAD_MUTEX_ERRORCHECK`, и все проверенные мной реализации не осуществляют такой проверки. Таким образом, борьба с мертвыми блокировками, включающими несколько нитей, оказывается обязанностью программиста.

Два основных способа борьбы с мертвой блокировкой - это упорядочение блокировок и групповые захваты мутексов. Более радикальные способы включают в себя запрет удерживать в каждый момент времени более одного мутекса и полный отказ от разделяемых данных и переход к взаимодействию через трубы или очереди

сообщений. Первый способ слишком радикален для большинства практических применений; второй способ лишает нас основного преимущества многопоточных программ (возможности взаимодействовать через разделяемую память) и не будет рассматриваться в этом курсе. Средств группового захвата мутексов POSIX Thread API не предоставляет, хотя такой захват можно реализовать самостоятельно, используя `pthread_mutex_trylock(3C)` и условные переменные (условные переменные рассматриваются в следующей лекции).

Оба эти способа страдают тем недостатком, что они нарушают требования модульности и инкапсуляции. При работе с программным модулем мы должны знать обо всех блокировках, которые этот модуль может установить. Особенно важен этот недостаток при использовании групповых блокировок. Упорядочение блокировок меньше страдает от этого недостатка.

Действительно, большинство реальных приложений и библиотек имеют многослойную структуру; при этом функции и методы "верхних" слоев вызывают функции и методы "нижних" слоев, но вызовы в обратном направлении происходят редко или вообще не происходит. Такая структура приложения создает естественный порядок захвата блокировок, и все, что требуется от программиста - не нарушать этот порядок умышленно, например, удерживая какую-то блокировку между вызовами функций своего слоя.

Впрочем, ряд приемов программирования, таких, как хуки, `callback`, сигналы, при определенных обстоятельствах - виртуальные методы и обработка исключений - приводят к нарушению порядка межслойного взаимодействия и, соответственно, могут привести к нарушению порядка захвата блокировок. Проявление этой проблемы мы видели на предыдущей лекции, когда обсуждали сигналы и атрибут Async-Signal-Safe.

Атрибуты мутексов

Как и `pthread_attr_t`, `pthread_mutex_attr_t` представляет собой непрозрачный тип данных, содержащий набор атрибутов мутекса. Для просмотра и изменения этих атрибутов необходимо использовать

get/set функции. Как и у pthread_attr_t, между атрибутами и мутексом не образуется никакой постоянной связи. Если завести переменную pthread_attr_t_attr и создать по неймутекс, последующие изменения attr неокажут никакого влияния на уже созданный мутекс. Большинство атрибутов мутекса вообще не могут быть изменены после создания мутекса.

Список атрибутов мутекса таков:

- pshared
- type
- protocol
- prioceiling
- robust_np (нестандартный)

Атрибут pshared определяет область действия мутекса. Допустимые значения - PTHREAD_PROCESS_SHARED (разделяемый мутекс, который может использоваться для межпроцессного взаимодействия) и PTHREAD_PROCESS_PRIVATE (локальный или приватный мутекс, пригодный только для синхронизации нитей одного процесса). Значение по умолчанию - PTHREAD_PROCESS_PRIVATE. Чтобы использовать разделяемый мутексдля межпроцессного взаимодействия, егонеобходимо разместить в сегменте разделяемой памяти, например в файле, отображенном на память с флагом MAP_SHARED, или в разделяемой памяти System V IPC.

Атрибут type обозначает способ проверки ошибок при работе с мутексом. Допустимые значения - PTHREAD_MUTEX_NORMAL, PTHREAD_MUTEX_ERRORCHECK, PTHREAD_MUTEX_RECURSIVE и PTHREAD_MUTEX_DEFAULT. Мутексы типа NORMAL не делают никаких проверок. Многократный захват мутекса одной нитью приводит к мертвой блокировке, результаты многократного освобождения мутекса или захвата мутекса одной нитью и освобождения другой не определены. На практике, в Solaris 10 и в Linux попытки освобождения свободного мутекса игнорируются, а попытка освобождения мутекса, захваченного другой нитью, приводит к тому, что мутекс переводится в состояние "свободен" и одна из заблокированных на нем нитей освобождается. Таким образом, в

некоторых реализациях мутексы типа NORMAL ведут себя скорее как двоичные семафоры, чем как мутексы в строгом смысле этого слова. Но стандарт POSIX не гарантирует, что мутексы будут вести себя именно таким образом во всех реализациях, в том числе и в будущих версиях Linux и Solaris.

Тип ERRORCHECK требует, чтобы все операции над мутексами проверяли состояние мутекса и возвращали ошибки при недопустимых последовательностях операций над мутексом. Из описания pthread_attr_settype(3C) можно сделать вывод, что такие мутексы делают проверку на мертвую блокировку с участием нескольких нитей, но это не так. В соответствии с требованиями стандарта, код ошибки EDEADLK возвращается только при попыткезахвата мутекса, уже занятого текущейнитью. Мутексытипа ERRORCHECK удобныдля отладки приложений, но требуют гораздо большего объема вычислений, чем мутексы типа NORMAL.

Мутексы типа RECURSIVE допускают многократный захват одной и той же нитью. Такой мутекс поддерживает счетчик захватов и должен быть освобожден нитью столько же раз, сколько был захвачен. Такие мутексы удобны для борьбы с мертвыми блокировками, возникающими в коде, аналогичном примеру 1. Попытка освобождениясвободного мутекса или мутекса, занятого другой нитью, приводит к ошибке.

Тип DEFAULT в соответствии со стандартом делает даже меньше проверок, чем тип NORMAL. Так, повторный захват мутекса типа NORMAL приводит к мертвой блокировке, а повторный захват мутекса типа DEFAULT, в соответствии со стандартом, может приводить к непредсказуемым последствиям вплоть до аварийного завершения процесса. Реализация имеет право отображать тип DEFAULT на любой другой тип. Solaris 10 отображает тип DEFAULT на тип NORMAL.

Атрибут protocol описывает схему предотвращения инверсии приоритета, используемую этим мутексом. Инверсия приоритета подробнее рассматривается в приложении к этой лекции. Это проблема, которая возникает, если низкоприоритетная нить удерживает мутекс, на котором ожидает высокоприоритетная нить. По умолчанию, мутексы создаются с протоколом PTHREAD_PRIO_NONE, т.е. не предоставляют никаких средств для борьбы с инверсией приоритета.

Два других допустимых значения `protocol` - это `PTHREAD_PRIO_INHERIT` и `PTHREAD_PRIO_PROTECT`. `INHERIT` обозначает наследование приоритета: приоритет нити, удерживающей мутекс, повышается до самого высокого из приоритетов нитей, ждущей этого мутекса. `PROTECT` обозначает так называемый "потолок приоритета" (priority ceiling): приоритет процесса, удерживающего мутекс, равен приоритету, указанному в свойствах этого мутекса, а именно - в свойстве `prioceiling`. Приоритет мутекса вообще говоря не зависит от приоритетов нитей, ждущих на этом мутексе. Если нить удерживает несколько мутексов с протоколами `INHERIT` и `PROTECT`, ее приоритет будет равен наивысшему из приоритетов, обеспечиваемыхэтими мутексами.

Атрибут `prioceiling` используется мутексами с протоколом `PTHREAD_PRIO_PROTECT`. При остальных значениях атрибута `protocol` этот атрибут игнорируется.

Атрибут `robust_np` используется главным образом с межпроцессными мутексами (`pshared==PROCESS_SHARED`). Это нестандартный атрибут, реализованный в Solaris; в Linux этот атрибут появился в версии ядра 2.6 и NPTL; большинство других реализаций POSIX Threads API его не поддерживают. Суффикс _np у имени атрибута сигнализирует, что этот атрибут не является частью стандарта POSIX. Как ни странно, в соответствии со страницей `pthread_attr_setrobust_np(3C)` системного руководства Solaris 10, для использования атрибута `robust_np` необходимо установить атрибут `protocol` равным `POSIX_PRIO_INHERIT`, а `pshared` может принимать любое значение.

Атрибут `robust_np` управляет поведением мутекса в ситуации, когда владелец этого мутекса аварийно завершился. Под аварийным завершениемподразумевается не `pthread_cancel(3C)`, а завершение процесса по сигналу. Завершение процесса по сигналу приводит к завершению всех нитей этого процесса; локальные мутексы и защищаемые ими ресурсы при этом уничтожаются и их судьба никого не интересует. Однако при использовании разделяемой памяти и разделяемых мутексов возможна ситуация, когда один из процессов завершается, а остальные продолжают работу. Если одна из нитей

завершившегося процесса удерживала мутекс, то нити остальных процессов, пытающиеся захватить этот мутекс, могут остаться в состоянии бесконечного ожидания. Эта проблема не может быть решена за счет принудительного освобождения мутекса, ведь занятость мутекса сигнализирует, что защищаемый им ресурс, скорее всего, находится в несогласованном состоянии.

Атрибут `robust_np` может принимать два значения: `PTHREAD_MUTEX_STALLED_NP` и `PTHREAD_MUTEX_ROBUST_NP`. Значение по умолчанию - `PTHREAD_MUTEX_STALLED_NP`. При этом мутексы, удерживавшиеся завершившимся процессом, просто остаются в занятом состоянии, и все нити, пытающиеся захватить их, блокируются.

В режиме *`ROBUST`* (надежный) мутекс реализует более сложное поведение. Первая нить, пытающаяся захватить мутекс, получает ошибку `EOWNERDEAD`, но мутекс при этом захватывается. Нить должна попытаться привести ресурс, защищаемый мутексом, в согласованное состояние. Если это удается, нить должна вызвать функцию `pthread_mutex_consistent_np(3C)`. После этого остальные нити смогут продолжить нормальную работу с мутексом и ресурсом. Если восстановить согласованное состояние ресурса не удается, нить должна просто освободить мутекс. После этого все попытки захватить такой мутекс будут завершаться с кодом ошибки `ENOTRECOVERABLE`.

Приложение. Инверсия приоритета и борьба с ней

Гэндальф: Это Балрог! Бежим. Гимли: Но мы всеравноне можем бежатьбыстрее демона! Леголас: Нам достаточно бежать быстрее тебя!

Анонимная пародия

В системах с приоритетным планированием при взаимодействии процессов и нитей с разными приоритетами возникает ряд специфических проблем, объединяемых названием инверсия приоритета (priority inversion).

Один из наглядных примеров этой проблемы можно наблюдать, если редактировать большой документ в текстовом процессоре Microsoft Word 2000 с включенным автосохранением. Проблема воспроизводится только если вставлять текст в начало или в середину документа, вынуждая Word переразбивать текст на страницы. При сохранении документа Word должен завершить разбиение на страницы, а для этого он вынужден остановить редактирование, иначе есть риск, что переразбиение на страницы никогда не завершится. Таким образом, при редактировании большого документа с включенным автосохранением пользователь большую часть времени взаимодействует с высокоприоритетным и действительно быстро реагирующим потоком, отвечающим за редактирование - но иногда оказывается вынужден ждать завершения работы низкоприоритетного фонового потока. Это выглядит как внезапное "зависание" Word. Если в интерактивных приложениях инверсия приоритета только раздражает, в системах жесткого реального времени она может приводить к действительно серьезным проблемам.

При расчете времени реакции на событие, разработчик системы реального времени должен принимать во внимание не только время исполнения кода, непосредственно обрабатывающего это событие, но и времена работы всех критических секций во всех нитях, которые могут удерживать мутексы, необходимые обработчику - ведь обработчик не сможет продолжить исполнение, пока не захватит эти мутексы, а произвольно снимать их нельзя, потому что они сигнализируют, что защищаемый ими разделяемый ресурс находится в несогласованном состоянии.

Если высокоприоритетная нить пытается захватить мутекс, занятый низкоприоритетной нитью, то в определенномсмысле получится, что эта нить будет работать со скоростью низкоприоритетного процесса.

В условиях, когда планировщик не обеспечивает справедливого распределения времени центрального процессора, а в системе наравне с высокоприоритетной нитью (работа которой нас интересует) и низкоприоритетной (которая держит мутекс) существуют еще среднеприоритетные нити, может оказаться, что низкоприоритетная нить не будет получать управления в течение значительного времени. На это же время будет заблокирована и высокоприоритетная нить.

Особенно серьезна эта проблема, когда высоко- и низкоприоритетная нити относятся к разным классам планирования - а в системах реального времени так оно и есть.

Инверсия приоритета в Mars Pathfinder

При разработке ПО для бортового компьютера космического аппарата Mars Pathfinder, была принята архитектура, которую сами разработчики называли "общей шиной" - глобальная разделяемая область памяти, которую все процессы в системе использовали для коммуникации. Доступ к этой области защищался мутексом.

В системе существовала высокоприоритетная нить, занимавшаяся управлением шиной, которая периодически запускалась по сигналам таймера и собирала из "шины" некоторую важную для нее информацию. Эта же нить должна была сбрасывать сторожевой таймер, подтверждая, что система функционирует нормально.

Кроме того, в системе существовало еще несколько нитей, осуществлявших доступ к шине, в частности нить сбора метеорологических данных. Эта нить также запускалась по расписанию и копировала данные в буфер "шины", разумеется, захватывая при этом мутекс.

Таким образом, если нить управления "шиной" запускалась во время работы "метеорологической" нити, то она должна была ждать некоторое дополнительное время. В моменты, когда в системе не было других активных нитей, это не приводило к проблемам.

Однако когда накладывалось исполнение трех нитей, интервал ожидания сильно увеличивался, и это приводило к срабатыванию сторожевого таймера и сбросу бортового компьютера - в данном случае, ложному, потому что система все-таки оставалась работоспособной.

При тестировании на Земле сбросы такого рода несколько раз происходили, но разработчики не смогли воспроизвести условия их возникновения (срабатывание ошибки требует наложения трех нитей, каждая из которых сама по себе большую часть времени заблокирована)

и отправили на Марс аппарат с недиагностированной проблемой. При эксплуатации непредсказуемые и происходящие, в среднем, раз в несколько дней сбросы бортового компьютера доставляли много неудобств центру управления полетом; к тому же, специалисты центра вынуждены были объяснять происходящее журналистам. На копии системы на Земле сбросы долгое время не удавалось воспроизвести; наконец, под утро, когда в лаборатории оставался только один разработчик, сброс все-таки произошел и по анализу отладочного дампа системы проблему удалось решить.

Основным средством борьбы с инверсией приоритета является наследование приоритета (priority inheritance). Обычно наследование контролируется флагом в параметрах мутекса или в параметрах системного вызова, захватывающего мутекс. Если высокоприоритетная нить пытается захватить мутекс с таким флагом, то приоритет нити, удерживающей этот мутекс, приравнивается приоритету нашей нити. Таким образом, в каждый момент времени, реальный приоритет нити, удерживающей мутекс, равен наивысшему из приоритетов нитей, ожидающих этого мутекса.

Более радикальное решение называется потолком приоритета (priority ceiling). Оно состоит в том, что приоритет нити, удерживающей мутекс, приравнивается наивысшему из приоритетов нитей, которые могут захватить этот мутекс. Разумеется, во время исполнения определить потолок приоритета невозможно, он должен устанавливаться программистом как параметр мутекса.

Легко понять, что во время работы в критических секциях, защищенных такими мутексами, задачи - даже низкоприоритетные - должны подчиняться всем ограничениям, которые может накладывать исполнение в режиме реального времени. Если мы в каком-то смысле не доверяем низкоприоритетной нити (в частности, не можем оценить время, в течение которого она будет удерживать мутекс), нам следует отказаться от использования разделяемой памяти для взаимодействия с этой нитью и перейти к каким-либо буферизованным средствам обмена данными. При этом достаточно обеспечить гарантированное время передачи данных в буфер; для того, чтобы избежать переполнения буфера, достаточно того, чтобы средний темп генерации данных высокоприоритетной нитью был ниже среднего же возможного темпа

их обработки низкоприоритетным потребителем. Впрочем, если темп такой обработки подвержен большим флуктуациям (например, из-за наличия в системе других процессов), может оказаться необходимо предусмотреть возможность сброса части данных для борьбы с переполнением буфера - именно так борются с перегрузками сетевые маршрутизаторы и коммутаторы.

Необходимо отметить, что и наследование, и потолок приоритета применимы только к мутексам и другим примитивам синхронизации, для которых можно указать, какая именно нить их удерживает. Точнее говоря, для борьбы с инверсией приоритета необходимо, чтобы приоритет повышался для нити, которая будет освобождать семафор. Для мутексов это всегда та же нить, которая его захватывала, но для семафоров-счетчиков это, вообще говоря, неверно.

Поэтому даже те системы, которые предоставляют наследование или потолки приоритетов для мутексов, не предоставляют аналогичных средств для семафоров-счетчиков. В книге [QNX 2004] приводятся результаты экспериментов над системой реального времени QNX, из которых видно, что эта ОС реализует наследование приоритетов на мутексах, но при использовании семафоров-счетчиков можно получить классический случай инверсии приоритета.

Блокировки чтения-записи, условные переменные, барьеры и семафоры-счетчики

В ходе этой лекции вы изучите использование следующих примитивов взаимоисключения и синхронизации: блокировок чтения-записи, условных переменных, барьеров, семафоров-счетчиков , блокировки чтения-записи.

Не надо звонить, мы поймаем машину внизу

Надеюсь, что ты разбудишь меня

Не раньше, чем нас довезут

Б. Гребенщиков

Блокировки чтения-записи (read-write locks) похожи на мутексы, но отличаются от них тем, что имеют два режима захвата - для чтения и для записи. Блокировку для чтения могут удерживать несколько нитей одновременно. Блокировку для записи может удерживать только одна нить; при этом никакая другая нить не может удерживать эту же блокировку для чтения. API для работы с блокировками чтения-записи в целом похож на API для работы с мутексами и включает в себя следующие функции:

- `pthread_rwlock_init(3C)`
- `pthread_rwlock_rdlock(3C)`
- `pthread_rwlock_tryrdlock(3C)`
- `pthread_rwlock_timedrdlock(3C)`
- `pthread_rwlock_wrlock(3C)`
- `pthread_rwlock_trywrlock(3C)`
- `pthread_rwlock_timedwrlock(3C)`
- `pthread_rwlock_unlock(3C)`
- `pthread_rwlock_destroy(3C)`
- `pthread_rwlockattr_init`
- `pthread_rwlockattr_destroy`
- `pthread_rwlockattr_getpshared(3C)`
- `pthread_rwlockattr_setpshared(3C)`

Набор атрибутов `pthread_rwlockattr_t` существенно беднее, чем у мутекса, и включает только атрибут `pshared`, управляющий областью действия блокировки. Применение блокировок чтения-записи достаточно очевидно. Они используются для защиты структур данных, которые читают значительно чаще, чем модифицируют.

Условные переменные

Условная переменная представляет собой примитив синхронизации. Над ней определены две основные операции: `wait` и `signal`. Нить, выполнившая операцию `wait`, блокируется до того момента, пока другая нить не выполнит операцию `signal`. Таким образом, операцией `wait` первая нить сообщает системе, что она ждет выполнения какого-то условия, а операцией `signal` вторая нить сообщает первой, что параметры, от которых зависит выполнение условия, возможно, изменились.

Основное применение условных переменных - это сценарий производитель-потребитель. Рассмотрим две нити, одна из которых генерирует данные,а другая - перерабатывает их. В простейшем случае производитель помещает каждую следующую порцию данных в разделяемую переменную, а потребитель считывает ее оттуда. При этом могут возникать две проблемы. Если производитель работает быстрее потребителя, то он может записать очередную порцию данных до того, как потребитель прочитает предыдущую. При этом предыдущая порция данных будет потеряна. Если же потребитель работает быстрее производителя, он может обработать одну и ту же порцию данных несколько раз. Легко понять, что при помощи одного мутекса эти проблемы устранить невозможно. Несколько сложнее понять, что их нельзя решить и при помощи двух мутексов (в качестве упражнения слушателю предлагается доказать это утверждение). Даже при использовании трех мутексов мы будем вынуждены использовать для начальной синхронизации холостой цикл или какой-то другой примитив межпоточного взаимодействия.

Классическое решение этой задачи реализуется с использованием условной переменной (см. пример 7.1).

```
int full;
pthread_mutex_t mx;
pthead_cond_t cond;
int data;
void *producer(void *) {
  while(1) {
    int t=produce();
    pthread_mutex_lock(&mx)
    while(full) {
      pthread_cond_wait(&cond, &mx);
    }
    data=t;
    full=1;
    pthread_cond_signal(&mx);
    pthread_mutex_unlock(&mx);
  }
  return NULL;
}

void * consumer(void *) {
  while (1) {
    int t;
    pthread_mutex_lock(&mx);
    while (!full) {
      pthread_cond_wait(&cond, &mx);
    }
    t=data;
    full=0;
    pthread_cond_signal(&mx);
    pthread_mutex_unlock(&mx);
    consume(1);
  }
  return NULL;
}
```

Пример 7.1. Решение задачи "производитель-потребитель" (фрагмент)

Условную переменную можно также использовать для реализации группового (атомарного) захвата нескольких мутексов (см. пример 7.2)

```
void get_forks (int phil, int fork1, int fork2) {
  int res;
  pthread_mutex_lock(&getting_forks_mx);
  do {
    if (res=pthread_mutex_trylock(&forks[fork1])) {
      res=pthread_mutex_trylock(&forks[fork2]);
      if (res) pthread_mutex_unlock(&forks[fork1]);
    }
    if (res) pthread_cond_wait(&getting_forks_cond, &getting_forks_mx);
  } while(res);
  pthread_mutex_unlock(&getting_forks_mx);
}

void down_forks (int f1, int f2) {
  pthread_mutex_lock(&getting_forks_mx);
  pthread_mutex_unlock (&forks[f1]);
  pthread_mutex_unlock (&forks[f2]);
  pthread_cond_broadcast(&getting_forks_cond);
  pthread_mutex_unlock(&getting_forks_mx);
}
```

Пример 7.2. Реализация атомарного захвата нескольких мутексов

Точное описание действия функции `pthread_cond_wait(3C)` звучит так. Эта функция имеет два параметра, `pthread_cond_t *cond` и `pthread_mutex_t *mx`. При вызове `wait` мутекс должен быть захвачен, в противном случае результат не определен. `Wait` освобождает мутекс и блокирует нить до момента вызова другой нитью `pthread_cond_signal`. После пробуждения `wait` пытается захватить мутекс; если это не получается, он блокируется до того момента, пока мутекс не освободят.

Мутекс используется для защиты данных, используемых при вычислении условия, с которым связана наша условная переменная. Условие необходимо проверять как перед вызовом `pthread_cond_wait(3C)`, так и после выхода из этой функции. Проверка условия перед вызовом позволяет защититься от так называемой "ошибки потерянного пробуждения" (lost wakeup), т.е. от

ситуации, когда производитель вызвал `signal` в то время, когда потребитель еще не был заблокирован в `wait`.

Интересный вопрос, обсуждающийся во многих документах - это следует ли вызывать `pthread_cond_signal(3C)`, удерживая мутекс `mx`, или нет. Сама по себе функция `pthread_cond_signal(3C)` нормально работает даже если нить не удерживает мутекс. Однако удержание мутекса `mx` (того же мутекса, который используется в параметрах `pthread_cond_wait(3C)`) гарантирует нас от посылки сигнала в момент проверки ждущей нитью условия или в интервале между проверкой условия и входом в `wait`. Таким образом, сигнализирование при свободном мутексе может привести к ошибке потерянного пробуждения.

Необходимость в проверке условия после выхода из `wait` не столь очевидна.

Действительно, решение задачи производитель-потребитель с одним производителеми одним потребителем будет работать и без этого. Но если производителей или потребителей будет несколько, между ними будет возникать конкуренция, и как раз для разрешения этой конкуренции необходимы мутекс и повторная проверка условия.

Полный список операций, определенных над условной переменной, таков:

- `pthread_condattr_destroy(3C)` - уничтожение атрибутов условной переменной
- `pthread_condattr_getpshared(3C)` - получение значения атрибута `pshared`
- `pthread_condattr_init(3C)` - инициализация атрибутов условной переменной
- `pthread_condattr_setpshared(3C)` - установка значения атрибута `pshared`
- `pthread_cond_broadcast(3C)` - широковещательный вариант операции `signal`
- `pthread_cond_destroy(3C)` - уничтожение условной переменной

- `pthread_cond_init(3C)` - инициализация условной переменной
- `pthread_cond_reltimedwait_np(3C)` - ожидание с тайм-аутом
- `pthread_cond_signal(3C)` - операция `signal`
- `pthread_cond_timedwait(3C)` - ожидание с тайм-аутом
- `pthread_cond_wait(3C)` - ожидание условной переменной (операция `wait`)

Процедура создания и уничтожениясловных переменных в целом аналогична процедуре создания и уничтожения ранее рассматривавшихся объектов синхронизации.

Барьеры

Барьер (*`barrier`*) - примитив синхронизации, применяемый главным образом в вычислительных задачах MPI/OpenMP. Существуют также примеры его применения в других задачах, но как правило это также задачи вычислительного характера. При создании барьера необходимо указать количество нитей `N`, необходимое для перехода через барьер. Нити, подходящие к барьеру, вызывают функцию `pthread_barrier_wait(3C)`. Если количество нитей, ожидающих возле барьера, меньше `N-1`, нить блокируется. Когда набирается `N` нитей, все они разблокируются и продолжают исполнение.

При разблокировании в одной из нитей `pthread_barrier_wait(3C)` возвращает значение `PTHREAD_BARRIER_SERIAL_THREAD`, а в остальных нитях - 0. Рассмотрим расчет видеокадра многопоточной программой на многопроцессорной машине. Нити, завершившие расчет своей части изображения, блокируются на барьере. Когда расчет изображения завершен, одна из нитей (получившая `PTHREAD_BARRIER_SERIAL_THREAD`) переключает видеоконтекст, то есть показывает изображение пользовтаелю, и нити продолжают расчеты.

Барьеры полезны для синхронизации нитей, выполняющих части одной

и той же работы, например параллельные вычисления. Они позволяют гарантировать, что даже при неравномерной загрузке системы нити будут выполнять равный объем работы в единицу времени.

Семафоры-счетчики

Исторически, семафоры-счетчики были одним из первых примитивов синхронизации. В старых учебниках они известны под названием "семафоры Дийкстры". Семафор представляет собой целочисленную переменную, над которой определены две операции, `post` и `wait`.

Операция `wait` пытается вычесть единицу из флаговой переменной семафора. Если значение флаговой переменной больше `0`, то происходит обычное вычитание и нить, выполнившая операцию `wait`, продолжает исполнение. Если значение флаговой переменной было равно `0`, то `wait` блокируется до тех пор, пока кто-то (скорее всего, другая нить) не увеличит значение этой переменной.

Операция `post` добавляет единицу к флаговой переменной семафора. Если при этом на семафоре ждала нить, операция `post` пробуждает эту нить.

Семафоры можно использовать в качестве примитивов взаимоисключения, при этом операция `wait` аналогична захвату мутекса, а операция `post` - операции его освобождения. Однако, в отличие от мутексов, операции `post` и `wait` не обязаны выполняться одной и той же нитью и даже не обязаны быть парными. Это позволяет использовать семафоры в различных ситуациях, которые сложно или невозможно разрешить при помощи мутексов. Иногда семафоры используют в качестве разделяемых целочисленных переменных, например в качестве счетчиков записей в очереди.

Широко известно решение задачи производитель-потребитель на двух семафорах (см. пример 7.3).

```
sem_t p, q;
int data;
void *producer(void *) {
```

```
  while(1) {
    int t=produce();
    sem_wait(&p);
    data=t;
    sem_post(&q);
   }
  return NULL;
}

void * consumer(void *) {
  while (1) {
    int t;
    sem_wait(&q);
    t=data;
    sem_post(&p);
    consume(1);
   }
  return NULL;
}
```

Пример 7.3. Решение задачи производитель-потребитель на двух семафорах

Семафоры также можно использовать для обхода проблемы инверсии приоритета. Если высокоприоритетная и низкоприоритетная нить должны взаимодействовать, иногда удается реализовать соглашение, что только низкоприоритетная нить может выполнять над семафором операцию `wait`, а высокоприоритетная нить может делать только `post`. Такое взаимодействие обычно похоже на схему производитель-потребитель с тем отличием, что производитель может терять некоторые порции данных, если потребитель за ним не успевает. При этом условии в примере 7.3 можно избавиться от семафора p.

Такое решениедовольно часто используется в системах жесткого реального времени, поэтому семафоры-счетчики считаются частью стандарта POSIX Real Time Extension, а не основного подмножества стандарта POSIX.

В Solaris функции работы с семафорами-счетчиками включены в библиотеку librt.so. Их использование требует сборки программы с

ключом -lrt. В отличие от остальных функций POSIX Thread API, функции работы с семафорами и сам тип семафора не имеют префикса `pthread_`.

Как и остальные примитивы взаимодействия, рассматривавшиеся ранее, семафор POSIX представляет собой непрозрачный тип данных, операции над которым должны осуществляться специальными функциями. Этот тип называется `sem_t` и определен в файле `<semaphore.h>`.

Семафоры бывают двух типов - именованные и неименованные. Те и другие семафоры хранятся в переменных типа `sem_t`, но процедура инициализации и уничтожения этих переменных отличается.

Неименованные семафоры инициализируются функцией `sem_init(3RT)`. Эта функция имеет три параметра:

- `sem_t * sem` - инициализируемый семафор
- `int pshared` - `0` если семафор будет локальным в пределах процесса, ненулевое значение - если семафор будет разделяемым между процессами
- `unsigned intvalue` - начальноезначение флаговой переменной семафора.

После работы семафор необходимо уничтожить функцией `sem_destroy(3RT)`.

Над семафорами определены операции `sem_post(3RT)`, `sem_wait(3RT)`, `sem_trywait(3RT)`, `sem_timedwait(3RT)` и `sem_getvalue(3RT)`. Операция `sem_getvalue(3RT)`, как сказано в системном руководстве, "получает значение семафора в некоторый неопределенный момент времени". В любом случае, очевидно, что в интервале между исполнением `sem_getvalue(3RT)` и проверкой значенияфлаговая переменная семафора может измениться. Поэтому тот факт, что `sem_getvalue(3RT)` вернул ненулевое значение, не означает, что вызов `sem_wait(3RT)` с этим семафором не будет заблокирован. `sem_getvalue` полезен главным образом в отладочных целях.

В соответствии со стандартом POSIX, если на семафоре ожидает одна или несколько нитей, `sem_getvalue(3RT)` вместо нуля может возвращать отрицательное значение, модуль которого равен количеству нитей. Это поведение не является обязательным и Solaris 10 этого не делает.

Именованные семафоры создаются функцией `sem_open(3RT)`. При помощи этой же функции можно получить доступ к уже существующему именованному семафору. Эта функция имеет два обязательных параметра и два необязательных:

- `const char * name` - имя семафора. Имя должно начинаться с символа ' / ' и не должно содержать других символов ' / '. Рекомендуется, чтобы имя не превышало 14 символов. В зависимости от реализации, объект с таким именем может либо появляться либо не появляться в корневом каталогекорневой файловой системы (в Solaris и Linux не появляется). В любом случае, для создания семафора не обязательно иметь право создания файлов в корневом каталоге.
- `int flags` - флаги. Может принимать значения `0`, `O_CREAT` и `O_CREAT|O_EXCL`, где `O_CREAT` и `O_EXCL` - константы, определенные в `<sys/fcntl.h>`. Смысл этих значений аналогичен соответствующим значениям флагов в параметрах `open(2)`. `0` означает попытку доступа к уже существующему семафору, `O_CREAT` - доступ к существующему семафору или попытку создания, если такого семафора нет, `O_EXCL` - ошибку, если при попытке создания обнаруживается, что такой семафор уже существует.
- `mode_t mode` - необязательный параметр, который используется, только если `flags` содержит бит `O_CREAT`. Обозначает права доступа к семафору, которые задаются девятибитовой маской доступа, похожей на маску доступа к файлам. Как и у файла, у семафора есть идентификаторы хозяина и группы. Идентификатор хозяина устанавливается равным эффективному идентификатору пользователя процесса, создавшего семафор, идентификатор группы - эффективному идентификатору группы процесса.
- `unsigned int value` - необязательный параметр, который

используется только если `flags` содержит бит `O_CREAT`. Содержит начальное значение флаговой переменной семафора при его создании.

Функция `sem_open(3RT)` возвращает указатель на семафор (`sem_t *`). При ошибке она возвращает нулевой указатель и устанавливает `errno`. Если процесс попытается несколько раз открыть один и тот же семафор, ему будут возвращать один и тот же указатель. Именованные семафоры всегда разделяемые между процессами. При доступе к существующему семафору проверяются права доступа по той же схеме, по которой в Unix системам проверяются права доступа к файлам. Для доступа к семафору процесс должен иметь права чтения и записи.

Для отсоединения от семафора и освобождения памяти из-под него необходимо использовать функцию `sem_close(3RT)`. Эта функция неявно вызывается при `exit(2)` и `exec(2)`. Однако закрытие именованного семафора процессом не прекращает существования семафора. Чтобы удалить семафор, необходимо вызвать функцию `sem_unlink(3RT)`. Это лишит новые процессы возможность видеть семафор как существующий (попытка исполнить `sem_open` с именем этого семафора без флага `O_CREAT` приведет к ошибке) и позволит создать новый семафор с тем же именем. Однако если в момент `sem_unlink(3RT)` один или несколько процессов работали с семафором, семафор продолжит свое существование до момента, пока все эти процессы не выполнят `sem_close(3RT)`. Исполнять `sem_unlink(3RT)` могут только владелец семафора и суперпользователь.

Во всем остальном именованный семафор не отличается от неименованного. Над ним можно выполнять те же операции, что и над неименованным, при помощи тех же самых функций.

Стандартного API для получения списка именованных семафоров, существующих в системе, не существует. Разумеется, полный список этих объектов существует вядре и может быть получен через API для доступа к структурам данных ядра, но эти API различаются в разных операционных системах.

Мультиплексирование ввода/вывода и асинхронный ввод/вывод

В ходе этой лекции вы изучите: использование системного вызова select, использование системного вызова poll, некоторые аспекты использования select/poll в многопоточных программах, стандартные средства асинхронного ввода/вывода.

Мы ждали его слишком долго

Что может быть глупее, чем ждать?

Б. Гребенщиков

Системный вызов select

Если ваша программа главным образом занимаетсяоперациями ввода/вывода, вы можете получить наиболее важные из преимуществ многопоточности в однопоточной программе, используя системный вызов `select(3C)`. В большинстве Unix-систем `select` является системным вызовом, или, во всяком случае, описывается в секции системного руководства 2 (системные вызовы), т.е. ссылка на него должна была бы выглядеть как `select(2)`, но в Solaris 10 соответствующая страница системного руководства размещена в секции `3C` (стандартная библиотека языка C).

Устройства ввода/вывода обычно работают гораздо медленнее центрального процессора, поэтому при выполнении операций с ними процессор обычно оказывается вынужден ждать их. Поэтому во всех ОС системные вызовы синхронного ввода/вывода представляют собой блокирующиеся операции.

Это относится и к сетевым коммуникациям - взаимодействие через Интернет сопряжено с большими задержками и, как правило, происходит через не очень широкий и/или перегруженный канал связи.

Если ваша программа работает с несколькими устройствами ввода/вывода и/или сетевыми соединениями, ей невыгодно блокироваться на

операции, связанной с одним из этих устройств, ведь в таком состоянии она может пропустить возможность совершить ввод/вывод с другого устройства без блокировки. Эту проблему можно решать при помощи создания нитей, работающих с различными устройствами. В предыдущих лекциях мы изучили все необходимое для разработки таких программ. Однако для решения этой проблемы есть и другие средства.

Системный вызов `select(3C)` позволяет ожидать готовности нескольких устройств или сетевых соединений (в действительности, готовности объектов большинства типов, которые могут быть идентифицированы файловым дескриптором). Когда один или несколько из дескрипторов оказываются готовы передать данные, `select(3C)` возвращает управление программе и передает списки готовых дескрипторов в выходных параметрах.

В качестве параметров `select(3C)` использует множества (наборы) дескрипторов. В старых Unix-системах множества были реализованы в виде 1024-разрядных битовых масок. В современныхUnix-системах и в других ОС, реализующих `select`, множества реализованы в виде непрозрачного типа `fd_set`, над которым определены некоторые теоретико-множественные операции, а именно - очистка множества, включение дескриптора в множество, исключение дескриптора из множества и проверка наличия дескриптора в множестве. Препроцессорные директивы для выполнения этих операций описаны на странице руководства `select(3C)`.

В 32-разрядных версиях Unix SVR4, в том числе в Solaris, `fd_set` по прежнему представляет собой 1024-битовую маску; в 64-разрядных версиях SVR4 это маска разрядности 65536 бит. Размер маски определяет не только максимальное количество файловых дескрипторов в наборе, но и максимальный номер файлового дескриптора в наборе. Размер маски в вашей версии системы можно определить во время компиляции по значению препроцессорного символа `FD_SETSIZE`. Нумерация файловых дескрипторов в Unix начинается с `0`, поэтому максимальный номер дескриптора равен `FD_SETSIZE-1`.

Таким образом, если вы используете `select(3C)`, вам необходимо установить ограничения на количество дескрипторов вашего процесса. Это может быть сделано шелловской командой `ulimit(1)` перед

запуском процесса или системным вызовом `setrlimit(2)` уже во время исполнения вашего процесса. Разумеется, `setrlimit(2)` необходимо вызвать до того, как вы начнете создавать файловые дескрипторы.

Если вам необходимо использовать более 1024 дескрипторов в 32-битной программе, Solaris 10 предоставляет переходный API. Для его использования необходимо определить препроцессорный символ `FD_SETSIZE` с числовым значением, превышающим 1024, перед включением файла `<sys/time.h>`. При этом в файле `<sys/select.h>` сработают необходимые препроцессорные директивы и тип `fd_set` будет определен как большая битовая маска, а `select` и другие системные вызовы этого семейства будут переопределены для использования масок такого размера.

В некоторых реализациях `fd_set` реализован другими средствами, без использования битовых масок. Например, Win32 предоставляет `select` в составе так называемого Winsock API. В Win32 `fd_set` реализован как динамический массив, содержащий значения файловых дескрипторов. Поэтому вам не следует полагаться на знание внутренней структуры типа `fd_set`.

Так или иначе, изменения размера битовой маски `fd_set` или внутреннего представления этого типа требуют перекомпиляции всех программ, использующих `select(3C)`. В будущем, когда архитектурный лимит в 65536 дескрипторов на процесс будет повышен, может потребоваться новая версия реализации `fd_set` и `select` и новая перекомпиляция программ. Чтобы избежать этого и упростить переход на новую версию ABI, компания Sun Microsystems рекомендует отказываться от использования `select(3C)` и использовать вместо него системный вызов `poll(2)`. Системный вызов `poll(2)` рассматривается далее на этой лекции.

Системный вызов `select(3C)` имеет пять параметров.

- `int nfds` - число, на единицу большее, чем максимальный номер файлового дескриптора во всех множествах, переданных как параметры.

- `fd_set *readfds` - Входной параметр, множество дескрипторов, которые следует проверять на готовность к чтению. Конец файла или закрытие сокета считается частным случаем готовности к чтению. Регулярные файлы всегда считаются готовыми к чтению. Также, если вы хотите проверить слушающий сокет TCP на готовность к выполнению `accept(3SOCKET)`, его следует включить в это множество. Также, выходной параметр, множество дескрипторов, готовых к чтению.
- `fd_set *writefds` - Входной параметр, множество дескрипторов, которые следует проверять на готовность к записи. Ошибка при отложенной записи считается частным случаем готовности к записи. Регулярные файлы всегда готовы к записи. Также, если вы хотите проверить завершение операции асинхронного `connect(3SOCKET)`, сокет следует включить в это множество. Также, выходной параметр, множество дескрипторов, готовых к записи.
- `fd_set *errorfds` - Входной параметр, множество дескрипторов, которые следует проверять на наличие исключительных состояний. Определение исключительного состояния зависит от типа файлового дескриптора. Для сокетов TCP исключительное состояние возникает при приходе внеполосных данных. Регулярные файлы всегда считаются находящимися в исключительном состоянии. Также, выходной параметр, множество дескрипторов, на которых возникли исключительные состояния.
- `struct timeval * timeout` - тайм-аут, временной интервал, задаваемый с точностью до микросекунд. Если этот параметр равен `NULL`, то `select(3C)` будет ожидать неограниченное время; если в структуре задан нулевой интервал времени, `select(3C)` работает в режиме опроса, то есть возвращает управление немедленно, возможно с пустыми наборами дескрипторов.

Вместо всех параметров типа `fd_set *` можно передать нулевой указатель. Это означает, что соответствующий класс событий нас не интересует. `select(3C)` возвращает общее количество готовых дескрипторов во всех множествах при нормальном завершении (в том числе при завершении по тайм-ауту), и `-1` при ошибке.

В примере 8.1 приводится использование `select(3C)` для копирования данных из сетевого соединения на терминал, а с терминала - в сетевое соединение. Эта программа упрощенная, она предполагает, что запись на терминал и в сетевое соединениеникогда не будет заблокирована. Поскольку и терминал, и сетевое соединение имеют внутренние буферы, при небольших потоках данных это обычно так и есть.

Пример 8.1 взят из книги У.Р. Стивенс, Unix: разработка сетевых приложений. Вместо стандартных системных вызовов используются "обертки", описанныев файле "`unp.h`"

```
#include "unp.h"


void str_cli(FILE *fp, int sockfd) {
    int maxfdp1, stdineof;
    fd_set rset;
    char sendline[MAXLINE], recvline[MAXLINE];

    stdineof = 0;
    FD_ZERO(&rset);
    for ( ; ; ) {
        if (stdineof == 0) FD_SET(fileno(fp), &rset);
        FD_SET(sockfd, &rset);
        maxfdp1 = max(fileno(fp), sockfd) + 1;
        Select(maxfdp1, &rset, NULL, NULL, NULL);
        if (FD_ISSET(sockfd, &rset)) { /* socket is readable */
            if (Readline(sockfd, recvline, MAXLINE) == 0) {
                if (stdineof == 1) return; /* normal termination */
                else err_quit("str_cli: server terminated prematurely");
                }
                Fputs(recvline, stdout);
        }
        if (FD_ISSET(fileno(fp), &rset)) { /* input is readable */
            if (Fgets(sendline, MAXLINE, fp) == NULL) {
                stdineof = 1;
                Shutdown(sockfd, SHUT_WR); /* send FIN */
                FD_CLR(fileno(fp), &rset);
```

```
                continue;
            }
            Writen(sockfd, sendline, strlen(sendline));
        }
    }
}
```

Пример 8.1. Двустороннее копирование данных между терминалом и сетевым соединением.

Обратите внимание, что программа примера 8.1 заново пересоздает множества дескрипторов перед каждым вызовом `select(3C)`. Это необходимо, потому что при нормальном завершении `select(3C)` модифицирует свои параметры.

`select(3C)` считается MT-Safe, однако при его использовании в многопоточной программе надо иметь в виду следующий момент. Действительно, сам по себе `select(3C)` не использует локальных данных и поэтому его вызов из нескольких нитей не должен приводить к проблемам. Однако если несколько нитей работают с пересекающимися наборами файловых дескрипторов, возможен такой сценарий:

1. Нить 1 включает дескриптор `s` в набор `readfds` и вызывает `select`.
2. `select` в нити `1` возвращает `s` как готовый для чтения
3. Нить 2 включает дескриптор `s` в набор `readfds` и вызывает `select`
4. `select` в нити 2 возвращает `s` как готовый для чтения
5. Нить 1 вызывает `read` из дескриптора `s` и получает все данные из его буфера
6. Нить 2 вызывает `read` из дескриптора `s` и блокируется.

Чтобы избежать этого сценария, работу с файловыми дескрипторами в таких условиях следует защищать мутексами или какими-то другими примитивами взаимоисключения.

Важно подчеркнуть, что защищать надо не `select`, а именно последовательность операций над конкретным файловым

дескриптором, начиная с включения дескриптора в множество для `select` и заканчивая приемом данных из этого дескриптора, точнее, обновлением указателей в буфере, в который вы считали эти данные. Если этого не сделать, возможны еще более увлекательные сценарии, например:

1. Нить 1 включает дескриптор `s` в набор `readfds` и вызывает `select`.
2. `select` в нити 1 возвращает `s` как готовый для чтения
3. Нить 2 включает дескриптор `s` в набор `readfds` и вызывает `select`
4. `select` в нити 2 возвращает `s` как готовый для чтения
5. Нить 1 вызывает `read` из дескриптора `s` и получает только часть данных из его буфера
6. Нить 2 вызывает `read` из дескриптора `s`, получает данные и записывает их поверх данных, полученных нитью 1

В лекции 10 мы рассмотрим архитектуру приложения, в котором несколько нитей работают с общим пулом файловых дескрипторов - так называемую архитектуру рабочих нитей (worker threads). При этом нити, разумеется, должны указывать друг другу, с какими именно дескрипторами они сейчас работают.

С точки зрения разработки многопоточных программ, важным недостатком `select(3C)` - или, возможно, недостатком POSIX Thread API - является тот факт, что примитивы синхронизации POSIX не являются файловыми дескрипторами и не могут использоваться в `select(3C)`. В то же время, при реальной разработке многопоточных программ, занимающихся вводом/выводом, часто было бы полезно ожидать в одной операции готовности файловых дескрипторов и готовности других нитей собственного процесса.

Мультиплексирование ввода при помощи poll(2)

Системный вызов `poll(2)` выполняет приблизительно те же задачи, что и `select(3C)`, но использует несколько более удобный способ передачи информации о том, какие дескрипторы его интересуют. `poll(2)` имеет три параметра:

- `struct pollfd` *`fds`*`[]` - массив описателей дескрипторов. Структура `pollfd` обсуждается далее в этом разделе
- `nfds_t nfds` - количество описателей в массиве *`fds`*
- `int timeout` - тайм-аут в миллисекундах. Если параметр `timeout` равен `0`, `poll` работает в режиме опроса (возвращает управление немедленно). Если он равен `-1`, `poll` ждет готовности дескрипторов неограниченное время.

`poll(2)` возвращает количество дескрипторов, с которыми произошли какие-то события, запрошенные программой либо представляющие интерес для нее. Если `poll(2)` возвращает управление по тайм-ауту, код возврата будет равен `0`. При ошибке `poll(2)` возвращает `-1` и устанавливает `errno`.

Структура `pollfd` имеет следующие поля:

- `int fd` - дескриптор файла. Если это поле имеет отрицательное значение, запись игнорируется.
- `short events` - события, связанные с `fd`, которые нас интересуют.
- `short revents` - `return events`, события, связанные с `fd`, которые реально произошли.

При вызове `poll` пользователь должен заполнить поля `fd` и `events` ; поле `revents` заполняется системным вызовом.

Поля `events` и `revents` представляют собой битовые маски, биты которых соответствуют типам событий. Вместо битов рекомендуется использовать символьные константы, определенные в `<poll.h>`

Основные используемые типы событий - `POLLIN` (проверять готовность к чтению), и `POLLOUT` (проверять готовность к записи). В действительности, эти типы композитные и представляют собой сочетания разных типов событий. Так, для сокетов TCP можно указывать проверку поступления внеполосных данных, для устройств STREAMS - проверку поступления приоритетных данных и т.д. В `revents` устанавливаются биты, соответствующие реально происшедшему событию, т.е. если вы заказывали ожидание `POLLIN`, не

обязательно в `revents` будут установлены все биты, входящие в маску `POLLIN`. Это необходимо иметь в виду при проверке `revents` (см. пример 8.2).

Кроме `POLLIN` и `POLLOUT`, в `revents` также могут появляться биты `POLLERR`, `POLLHUP` и `POLLNVAL`. В `events` эти биты игнорируются, а в `revents` могут быть установлены при следующих условиях:

- `POLLERR` - на устройстве возникла ошибка
- `POLLHUP` - сокет, труба или терминальное устройство закрыты на другом конце
- `POLLNVAL` - значение `fd` не соответствует валидному файловому дескриптору (скорее всего, дескриптор был закрыт на нашем конце).

```
#include <poll.h>
struct pollfd fds[3];
int ifd1, ifd2, ofd, count;



fds[0].fd = ifd1;
fds[0].events = POLLNORM;
fds[1].fd = ifd2;
fds[1].events = POLLNORM;
fds[2].fd = ofd;
fds[2].events = POLLOUT;
count = poll(fds, 3, 10000);
if (count == -1) {
    perror("poll failed");
    exit(1);
}
if (count==0)
    printf("No data for reading or writing\n");
if (fds[0].revents & POLLNORM)
    printf("There is data for reading fd %d\n", fds[0].fd);
if (fds[1].revents & POLLNORM)
    printf("There is data for reading fd %d\n", fds[1].fd);
```

```
if (fds[2].revents & POLLOUT)
    printf("There is room to write on fd %d\n", fds[2].fd);
```

Пример 8.2. Использование poll(2) (фрагмент программы)

Преимущества `poll(2)` перед `select(3C)` достаточно очевидны:

1. интерфейс `poll` не накладывает ограничений на пространство номеров дескрипторов, во всяком случае пока эти номера входят в диапазон представления `int`.
2. при большом пространстве номеров дескрипторов (`65536` в данном контексте следует считать большим пространством), `poll` часто требует передачи между пользовательским процессом и ядром меньшего объема данных, чем `select`.
3. `poll` сообщает больше информации о происшедших с дескриптором событиях, чем может сообщить `select`
4. У `poll` входные и выходные значения разнесены по разным полям структуры, так что не требуется полностью пересоздавать массив *`fds`* после каждого вызова.

При использовании `poll(2)` в многопоточной программе приложимы те же соображения, которые высказывались в конце предыдущего раздела.

Использование /dev/poll

Использование `poll(2)` с большим количеством файловых дескрипторов приводит к передаче больших объемов данных между пользовательским процессом и ядром. При этом, скорее всего, большая часть этих данных передается впустую - ведь если процесс действительно может работать с таким большим числом дескрипторов, это, скорее всего, означает, что большинство из них не готовы к работе.

В Solaris предоставляется нестандартный API, который может использоваться для решения этой проблемы. Этот API описывается на странице системного руководства `poll(7D)` и состоит в использовании специального псевдоустройства `/dev/poll`.

Это устройство открывается как обычный файл системным вызовом `open(2)`. Затем в него следует записать одну или несколько структур `pollfd` (т.е. тех же самых структур, которые использует `poll(2)`). Запись осуществляется системным вызовом `write(2)` и может осуществляться в несколько приемов. При этом, если вы несколько раз записываете структуры, соответствующие одному и тому же дескриптору, с разными значениями поля `events`, это будет означать расширение списка опрашиваемых событий для вашего дескриптора. Т.е. если вы сначала запишете `pollfd` с `events==POLLIN`, а затем с `events==POLLOUT`, дескриптор будет опрашиваться в режиме `POLLIN | POLLOUT`.

Если вы хотите исключить дескриптор из множества опрашиваемых, вам следует записать структуру `pollfd`, в которой поле `events` содержит бит `POLLREMOVE`.

Многократное открытие `/dev/poll` одним процессом приводит к созданию нескольких независимых наборов дескрипторов.

Сам опрос осуществляется вызовом `ioctl(2)` с командой `DP_POLL`. Этот `ioctl` использует в качестве параметра значение `struct dvpoll *`. Тип `struct dvpoll` описан в `<sys/devpoll.h>` и содержит следующие поля:

- `struct pollfd* dp_fds` - указатель на массив, в который следует положить описатели дескрипторов, с которым связаны события
- `int dp_nfds` - размер массива `dp_fds`. Также, максимальное количество описателей дескрипторов, которые следует получить
- `int dp_timeout` - тайм-аут в миллисекундах. Этот параметр соответствует параметру timeout poll(2).

`Ioctl DP_POLL` возвращает количество описателей файловых дескрипторов, записанных в `dp_fds`, `0` если `ioctl` был разблокирован по тайм-ауту и `−1` в случае ошибки.

С практической точки зрения, важное отличие этого API от `poll(2)` состоит в том, что при использовании `poll(2)` описатели

дескрипторов после опроса расположены на тех же местах в массиве, на которых вы сами их разместили. Напротив, `ioctl DP_POLL` возвращает вам массив, который включает только те дескрипторы, с которыми связаны события, причем эти дескрипторы лежат в массиве в том порядке, в котором их счел удобным разместить драйвер `/dev/poll`. Т.е. вы должны просматривать `dp_fds` в порядке увеличения индекса, используя затем значение поля `fd` как ключ поиска. Скорее всего, вам придется завести массив (возможно, ассоциативный), связывающий значение файлового дескриптора с метаинформацией о том, что это за дескриптор и что вы с ним хотели делать. При работе с этим массивом вам следует иметь в виду, что Unix при нормальной работе переиспользует номера файловых дескрипторов, т.е. вам надо обновлять данные в вашем массиве каждом закрытии файла.

Корректная реализация всей требуемой функциональности (особенно в многопоточной программе) требует большого объемакода, причем кода, довольно сложного при отладке. Скорее всего, именно поэтому поддержка `poll(7D)` приложениями ограниченна и этот API не стал ни юридическим стандартом, ни даже стандартом де-факто. Большинство современных Unix-систем, за исключением Solaris, не поддерживают /dev/poll и не имеют планов реализации такой поддержки в обозримом будущем.

Однако с точки зрения производительности `poll(7D)` дает ощутимые преимущества даже при вполне реалистичных количествах файловых дескрипторов на процесс. По данным измерений, опубликованных на сайте ссылка: http://developers.sun.com/solaris/articles/polling_efficient.html - http://developers.sun.com/solaris/articles/polling_efficient.html, при 4000 тысячах файловых дескрипторов, `poll(7D)` требует в 16 раз меньше процессорного времени на исполнение заданного количества циклов опроса-чтения-записи, чем `poll(2)`!

Асинхронный ввод/вывод

Еще одна альтернатива многопоточности для приложений, ориентированных на ввод/вывод - это асинхронный ввод/вывод.

Традиционно, Unix использует синхронную модель ввода/вывода. Системные вызовы `read(2)`, `write(2)` и их аналоги возвращают управление только того, как данные уже прочитаны или записаны. Часто это приводит к блокировке нити.

Примечание

В действительности, не все так просто. `read(2)` действительно должен ожидать физического прочтения данных с устройства, но `write(2)` по умолчанию работает в режиме отложенной записи: он возвращает управление после того, как данные переданы в системный буфер, но, вообще говоря, до того, как данные будут физически переданы устройству. Это, как правило, значительно повышает наблюдаемую производительность программы и позволяет использовать память изпод данных для других целей сразу после возврата `write(2)`. Но отложенная запись имеет и существенные недостатки. Главный из них - вы узнаете о результате физической операции не сразу по коду возврата `write(2)`, а лишь через некоторое время после возврата, обычно по коду возврата следующего вызова `write(2)`. Для некоторых приложений - для мониторов транзакций, для многих программ реального времени и др. - это неприемлемо и они вынуждены выключать отложенную запись. Это делается флагом `O_SYNC`, который может устанавливаться при открытии файла и изменяться у открытого файла вызовом `fcntl(2)`.

Синхронизация отдельных операций записи может быть обеспечена вызовом `fsync(2)`. Для многих приложений, работающих с несколькими устройствами и/или сетевыми соединениями синхронная модель неудобна. Работа в режиме опроса тоже не всегда приемлема. Дело в том, что `select(3C)` и `poll(2)` считают дескриптор файла готовым для чтения только после того, как в его буфере физически появятся данные. Но некоторые устройства начинают отдавать данные только после того, как их об этом явно попросят.

Также, для некоторых приложений, особенно для приложений реального времени важно знать точный момент начала поступления данных. Для таких приложений также может быть неприемлемо то, что `select(3C)` и `poll(2)` считают регулярные файлы всегда готовыми для чтения и записи. Действительно, файловая система читается с диска

и хотя она работает гораздо быстрее, чем большинство сетевых соединений, но все-таки обращения к ней сопряжены с некоторыми задержками. Для приложений жесткого реального времени эти задержки могут быть неприемлемы - но без явного запроса на чтение файловая система данные не отдает!

С точки зрения приложений жесткого реального времени может оказаться существенным еще один аспект проблемы ввода/вывода. Дело в том, что приложения жесткого РВ имеют более высокий приоритет, чем ядро, поэтому исполнение ими системных вызовов - даже неблокирующихся! - может привести к инверсии приоритета.

Решение этих проблем известно давно и называется асинхронный ввод/вывод. В этом режиме системные вызовы ввода/вывода возвращают управление сразу после формирования запроса к драйверу устройства, как правило, даже до того, как данные будут скопированы в системный буфер. Формирование запроса состоит в установке записи (IRP, Input/Output Request Packet, пакет запроса ввода вывода) в очередь. Для этого надо лишь ненадолго захватить мутекс, защищающий "хвост" очереди, поэтому проблема инверсии приоритета легко преодолима. Для того, чтобы выяснить, закончился ли вызов, и если закончился, то чем именно, и можно ли использовать память, в которой хранились данные, предоставляется специальный API (см. рис. 8.1)

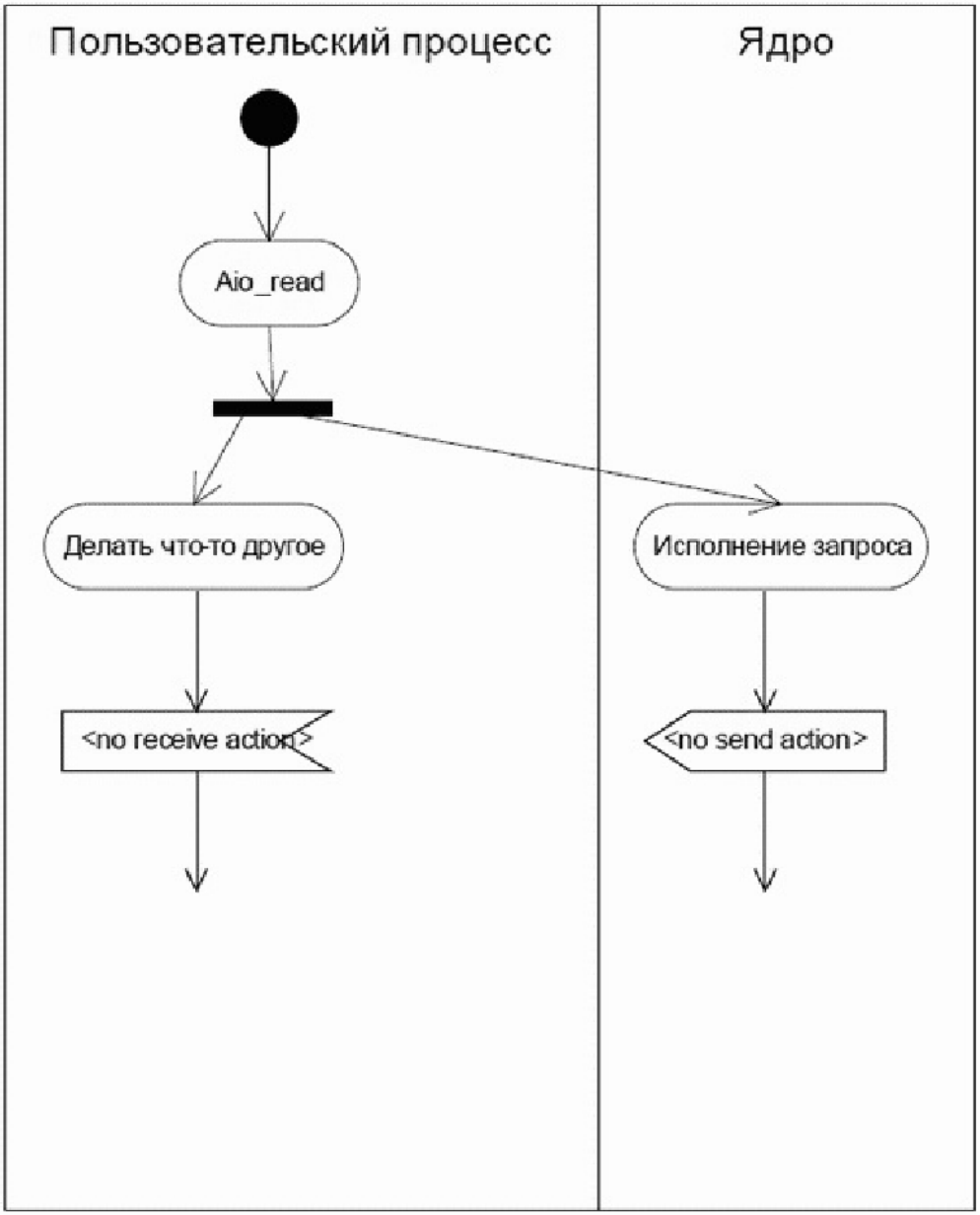


Рис. 8.1. Асинхронная модель ввода/вывода

Асинхронная модель была основной моделью ввода/вывода в таких ОС, как DEC RT-11, DEC RSX-11, VAX/VMS, OpenVMS. Эту модель в той или иной форме поддерживают практически все ОС реального времени. В системах семейства Unix с конца 1980х использовалось

несколько несовместимых API для асинхронного ввода/вывода. В 1993 году ANSI/IEEE был принят документ POSIX 1003.1b, описывающий стандартизованный API, который мы и изучим далее в этом разделе.

В Solaris 10 функции асинхронного ввода/вывода включены в библиотеку `libaio.so`. Для сборки программ, использующих эти функции, необходимо использовать ключ `-laio`. Для формирования запросов на асинхронный ввод/вывод используются функции `aio_read(3AIO)`, `aio_write(3AIO)` и `lio_listio(3AIO)`.

Функции `aio_read(3AIO)` и `aio_write(3AIO)` имеют единственный параметр, `structaiocb *aiocbp`. Структура `aiocb` определена в файле `<aio.h>` и содержит следующие поля:

- `int aio_fildes` - дескриптор файла
- `off_t aio_offset` - смещение в файле, начиная с которого будет идти запись или чтение
- `volatile void* aio_buf` - буфер, в который следует прочитать данные или в котором лежат данные для записи.
- `size_t aio_nbytes` - размер буфера. Как и традиционный `read(2)`, `aio_read(3AIO)` может прочитать меньше данных, чем было запрошено, но никогда непрочитает больше.
- `int aio_reqprio` - приоритет запроса
- `struct sigevent aio_sigevent` - способ оповещения о завершении запроса (рассматривается далее в этом разделе)
- `int aio_lio_opcode` - при `aio_read(3AIO)` и `aio_write(3AIO)` не используется, используется только функцией `lio_listio`.

Функция `lio_listio(3AIO)` позволяет одним системным вызовом сформировать несколько запросов на ввод/вывод. Эта функция имеет четыре параметра:

- `int mode` - может принимать значения `LIO_WAIT` (функция ждет завершения всех запросов) и `LIO_NOWAIT` (функция возвращает управление сразу после формирования всех запросов).
- `struct aiocb *list[]` - список указателей на структуры `aiocb` с описаниями запросов.

Запросы могут быть как на чтение, так и на запись, это определяется полем `aio_lio_opcode`. Запросы к одному дескриптору исполняются в том порядке, в каком они указаны в массиве `list`.

- `int nent` - количество записей в массиве `list`.
- `struct sigevent *sig` - способ оповещения о завершении всех запросов. Если `mode==LIO_WAIT`, этот параметр игнорируется.

Библиотека POSIX AIO предусматривает два способа оповещения программы о завершении запроса, синхронный и асинхронный. Сначала рассмотрим синхронный способ. Функция `aio_return(3AIO)` возвращает статус запроса. Если запрос уже завершился и завершился успешно, она возвращает размер прочитанных или записанныхданных в байтах. Как и у традиционного `read(2)`, в случае конца файла `aio_return(3AIO)` возвращает `0` байт. Если запрос завершился ошибкой или еще не завершился, возвращается `-1` и устанавливается `errno`. Если запрос еще не завершился, код ошибки равен `EINPROGRESS`.

Функция `aio_return(3AIO)` разрушающая; если ее вызвать для завершенного запроса, то она уничтожит системный объект, хранящий информацию о статусе запроса. Многократный вызов `aio_return(3AIO)` по поводу одного и того же запроса, таким образом, невозможен.

Функция `aio_error(3AIO)` возвращает код ошибки, связанной с запросом. При успешном завершении запроса возвращается `0`, при ошибке - код ошибки, для незавершенных запросов - `EINPROGRESS`.

Функция `aio_suspend(3AIO)` блокирует нить до завершения одного из указанных ей запросов асинхронного ввода/вывода либо на указанный интервал времени. Эта функция имеет три параметра:

- `const struct aiocb *const list[]` - массив указателей на описатели запросов.
- `int nent` - количество элементов в массиве `list`.

- `const struct timespec *timeout` - тайм-аут с точностью до наносекунд (в действительности, с точностью до разрешения системного таймера).

Функция возвращает `0`, если хотя бы одна из операций, перечисленныхв списке, завершилась. Если функция завершилась с ошибкой, она возвращает `-1` и устанавливает `errno`. Если функция завершилась по тайм-ауту, она также возвращает `-1` и `errno==EINPROGRESS`.

Пример использования асинхронного ввода/вывода с синхронной проверкой статуса запроса приводится в примере 8.3.

```
const char req[]="GET / HTTP/1.0\r\n\r\n";
int main() {
    int s;
    static struct aiocb readrq;
    static const struct aiocb *readrqv[2]={&readrq, NULL};

    /* Открыть сокет […] */
    memset(&readrq, 0, sizeof readrq);
    readrq.aio_fildes=s;
    readrq.aio_buf=buf;
    readrq.aio_nbytes=sizeof buf;
    if (aio_read(&readrq)) {
        /* … */
        }
        write(s, req, (sizeof req)-1);
        while(1) {
        aio_suspend(readrqv, 1, NULL);
        size=aio_return(&readrq);
        if (size>0) {
            write(1, buf, size);
            aio_read(&readrq);
        } else if (size==0) {
            break;
        } else if (errno!=EINPROGRESS) {
                perror("reading from socket");
        }
```

```
    }
  }
```

Пример 8.3. Асинхронный ввод/вывод с синхронной проверкой статуса запроса. Код сокращен, из него исключены открытие сокета и обработка ошибок.

Асинхронное оповещение приложения о завершении операций состоит в генерации сигнала при завершении операции. Чтобы это сделать, необходимо внести соответствующие настройки в поле `aio_sigevent` описателя запроса. Поле `aio_sigevent` имеет тип `struct sigevent`. Эта структура определена в `<signal.h>` и содержит следующие поля:

- `int sigev_notify` - режим нотификации. Допустимые значения - `SIGEV_NONE` (не посылать подтверждения), `SIGEV_SIGNAL` (генерировать сигнал при завершении запроса) и `SIGEV_THREAD` (при завершении запроса запускать указанную функцию в отдельной нити). Solaris 10 также поддерживает тип оповещения `SIGEV_PORT`, который рассматривается в приложении к этой лекции.
- `int sigev_signo` - номер сигнала, который будет сгенерирован при использовании `SIGEV_SIGNAL`.
- `union sigval sigev_value` - параметр, который будет передан обработчику сигнала или функции обработки. При использовании для асинхронного ввода/вывода это обычно указатель на запрос.

  При использовании `SIGEV_PORT` это должен быть указательна структуру `port_event_t`, содержащую номер порта и, возможно, дополнительные данные.

- `void (*sigev_notify_function)(union sigval)` - функция, которая будет вызвана при использовании `SIGEV_THREAD`.
- `pthread_attr_t *sigev_notify_attributes` - атрибуты нити, в которой будет запущена
- `sigev_notify_function` при использовании `SIGEV_THREAD`.

Далеко не все реализации `libaio` поддерживают оповещение `SIGEV_THREAD`. Некоторые Unix-системы используют вместо него нестандартное оповещение `SIGEV_CALLBACK`. Далее в этой лекции мы будем обсуждать только оповещение сигналом.

В качестве номера сигнала некоторые приложения используют `SIGIO` или `SIGPOLL` (в Unix SVR4 это один и тот же сигнал). Часто используют также `SIGUSR1` или `SIGUSR2` ; это удобно потому, что гарантирует, что аналогичный сигнал не возникнет по другой причине.

В приложениях реального времени используются также номера сигналов в диапазоне от `SIGRTMIN` до `SIGRTMAX`. Некоторые реализации выделяют для этой цели специальный номер сигнала `SIGAIO` или `SIGASYNCIO`, но в Solaris 10 такого сигнала нет.

Разумеется, перед тем, как исполнять асинхронные запросы с оповещением сигналом, следует установить обработчик этого сигнала. Для оповещения необходимо использовать сигналы, обрабатываемые в режиме `SA_SIGINFO`. Установить такой обработчик при помощи системных вызовов `signal(2)` и `sigset(2)` невозможно, необходимо использовать `sigaction(2)`. Установка обработчиков при помощи `sigaction` рассматривается в приложении 2 к этой лекции.

Обработка сигнала в режиме `SA_SIGINFO` имеет два свойства, каждое из которых полезно для наших целей. Во первых, обработчики таких сигналов имеют три параметра, в отличие от единственного параметра у традиционных обработчиков. Первый параметр имеет тип `int` и соответствует номеру сигнала, второй параметр имеет тип `siginfo_t *`, генерируется системой и содержит ряд интересных полей. Нас в данном случае больше всего интересует поле этой структуры `si_value`. Как раз в этом поле нам передается значение `aio_sigevent.sigev_value`, которое мы создали при настройке структуры `aiocb`.

Приложение 1. Порты Solaris

В Solaris 10 введен новый API, который может быть полезен и в

качестве замены `select/poll`, и для управления запросами асинхронного ввода/вывода. В действительности, `select` и `poll` в Solaris 10 реализуются с использованием портов.

Порт представляет собой объект, идентифицируемый файловым дескриптором. Порт создается функцией `port_create(3C)` и уничтожается системным вызовом `close(2)`. С уже созданным портом можно связывать объекты, способные генерировать события. В настоящее время поддерживаются следующие типы источников событий:

- `PORT_SOURCE_AIO` - запросы асинхронного ввода-вывода. Событием считается завершение запроса.
- `PORT_SOURCE_FD` - файловые дескрипторы. При ассоциации файлового дескриптора с портом необходимо указать флаги в формате поля `events` структуры `pollfd` ; событиеэти флаги и будут задавать события, которые нас интересуют.
- `PORT_SOURCE_TIMER` - таймеры.
- `PORT_SOURCE_USER` - события, генерируемые пользователем при помощи функции `port_send(3C)`.
- `PORT_SOURCE_ALERT` - события, генерируемые пользователем при помощи функции `port_send(3C)`.

Связывание асинхронного запроса с портомпроисходитвмомент создания запроса, установкой поля `aio_sigevent.sigev_notify=SIGEV_PORT`. Связывание файлового дескриптора с портом производится функцией `port_associate(3C)`. Судя по количеству и типам параметров, эта функция должна допускать связывание с портом различных объектов, но в Solaris 10 поддерживается единственный тип источника `PORT_SOURCE_FD`.

С одним портом можно связывать разнородные источники событий.

После того, как источники связаны с портом, можно получать информацию о возникших событиях функциями `port_get(3C)` и `port_getn(3C)`. После того, как порт вернул информацию о событии, источник события отсоединяется от порта. Если от этого

источника ожидаются какие-то еще события, источник следует связать с портом заново. В случае запросов асинхронного ввода/вывода и таймера смысл отсоединения очевиден. В случае с файлами смысл тоже достаточно легко понять - это защищает нас от неприятных сценариев использования `select/poll` в многопоточной программе, которые мы рассматривали в ходе этой лекции. При использовании традиционного `select/poll` существует возможность, что несколько нитей получат один и тот же файловый дескриптор и попытаются что-то с ним сделать. В лучшем случае это приведет к блокировке некоторых нитей, в худшем - к потере данных. Использование портов защищает нас от такого развития событий -после того, как одна из нитей получит файловый дескриптор у порта, он будет отсоединен и остальные нити не смогут получить его, пока он не будет присоединен снова явным образом.

Порты представляют собой новый API. В страницах системного руководства Solaris 10 функции этого API отмечены как `evolving` (развивающиеся). В настоящее время важным недостатком портов является то, что примитивы синхронизации POSIX Thread API нельзя использовать в качестве источников событий наравне с файловыми дескрипторами и таймерами.

Приложение 2. Установка обработчиков сигналов при помощи sigaction(2)

Системный вызов `sigaction(2)` дает ранее недоступный уровень управления сигналами. Кроме того, он совместим с требованиями стандарта POSIX. `sigaction(2)` предоставляет возможности, ранее реализованные в `sigset(2)`: закрывает окно уязвимости при использовании `signal(2)` и автоматически блокирует рекурсивный вызов функции обработки сигнала.

Параметры `act` и `oact`, если они не равны нулю, указывают на экземпляры `struct sigaction` со следующими полями:

- `void (*sa_handler)(int);` -Адрес традиционного обработчика сигнала, `SIG_IGN` или `SIG_DFL`
- `void (*sa_sigaction)(int, siginfo_t *, void`

`*)` - Адрес обработчика сигнала в режиме `SA_SIGINFO`. Реализации могут совмещать это поле с полем `sa_handler`, поэтому не следует пытаться установить `sa_handler` и `sa_sigaction` в одной структуре.

- `sigset_t sa_mask` -Маска сигналов, которые должны быть заблокированы, когда вызывается функция обработки сигнала.
- `int sa_flags` - Флаги, управляющие доставкой сигнала.

Если аргумент `act` ненулевой, он указывает на структуру, определяющую новые действия, которые должны быть предприняты при получении сигнала `sig`. Если аргумент `oact` ненулевой, он указывает на структуру, где сохраняются ранее установленные действия для этого сигнала.

Поле `sa_flags` в `struct sigaction` формируется побитовым ИЛИ следующих значений:

- `SA_ONSTACK` -Используется для обработки сигналов на альтернативном сигнальном стеке.
- `SA_RESETHAND` - Во время исполнения функции обработки сбрасывает реакцию на сигнал к `SIG_DFL` ; обрабатываемый сигнал при этом не блокируется.
- `SA_NODEFER` - Во время обработки сигнала сигнал не блокируется.
- `SA_RESTART` -Системные вызовы, которые будут прерваны исполнением функции обработки, автоматически перезапускаются.
- `SA_SIGINFO` -Используется для доступа к подробной информации о процессе, исполняющем сигнальный обработчик, такой как причина возникновения сигнала и контекст процесса в момент доставки сигнала.
- `SA_NOCLDWAIT` - Подавляет создание процессов-зомби.
- `SA_NOCLDSTOP` -Подавляет генерацию `SIGCHLD`, когда порожденные процессы останавливаются или возобновляются.

Событийно-ориентированные архитектуры

У многих современных программистов слова "событийно-ориентированная архитектура" ассоциируются в первую очередь с библиотеками для разработки графических пользовательских интерфейсов - AWT/Swing, Qt и др. В действительности, событийно-ориентированные архитектуры были изобретены гораздо раньше и совсем для других целей, только тогда еще никто не знал, что это так называется. Событийно-ориентированную архитектуру имеют ядра операционных систем (в том числе систем семейства Unix), многие серверные приложения и приложения реального времени. В ходе этой лекции вы изучите событийно-ориентированную архитектуру типичного серверного приложения и получите обзор других подходов к реализации и использованию событийно-ориентированных архитектур.

Обработка событий

Идею, лежащую в основе событийно-ориентированной архитектуры, можно описать следующим образом. программа рассматривается как набор объектов, реагирующих на события. События могут быть как внешними по отношению к системе - например, нажатие пользователем левой кнопки мыши, приход очередной порции данных из сетевого соединения, сработкой датчика в системе реального времени - так и внутренними. Внутренние события объекты-обработчики используют для коммуникации друг с другом.

Желая обрабатывать события определенного типа, программист регистрирует объект-обработчик.

При возникновении события, система вызывает метод объекта, ассоциированный с этим событием. Метод обрабатывает событие (возможно, генерируя при этом события, предназначенные другим объектам) и завершается. Если методы всех обработчиков выполняют свои операции быстро и без применения блокирующих системных вызовов, можно обеспечить высокую скорость реакции на события даже в рамках однопоточной программы. Если события поступают быстрее, чем обработчики успевают их обрабатывать, система выстраивает их в очередь.

В типичной системе с графическим пользовательским интерфейсом обработка событий происходит в несколько этапов. Рассмотрим простой пример окна с кнопкой "OK". Окна во всех системах графического интерфейса представляют собой обработчики событий. При добавлении кнопки в окно регистрируется еще один обработчик событий, связанный с кнопкой. Все, что делает этот обработчик при нажатии мышью на кнопку - генерирует командное событие, предназначенное окну. Далее,если наше окно представляет собой не главное окно приложения, а модальный диалог (например, диалог открытия файла), обработка командного события от кнопки "OK" может свестись к посылке командного события главному окну - в случае диалога открытия файла это командное событие должно содержать команду "открыть файл" с указанным именем файла.

Основные идеи, лежащие в основе современных графических интерфейсов, были разработаны во второй половине1970х в исследовательской лаборатории XeroxPARC. Однако сама по себе событийно ориентированная архитектура была придумана еще в 60е годы XX столетия и использовалась в ядрах операционных систем, главным образом при реализации подсистемы ввода-вывода. Драйвер устройства для типичной многозадачной ОС представляет собой набор функций ("точек входа") и блок переменных состояния устройства. Указатель на этот блок переменных состояния передается каждой из функций драйвера в качестве параметра. Благодаря этому, драйвер может обслуживать несколько однотипных устройств - ему для этого достаточно создать и зарегистрировать несколько блоков переменных состояния.

Фактически, драйвер представляет собой нечто очень похожее на объект в объектно-ориентированном программировании, однако в большинстве ОС драйверы до сих пор разрабатываются на не-объектно-ориентированных языках, чаще всегона C или даже на ассемблере. Наиболее известное исключение представляет ОС Apple Darwin (ядро MacOS X), в которой драйверы разрабатываются на специализированном подмножестве C++.

Драйвер обслуживает несколько потоков событий. Типичный драйвер физического устройства обслуживает два потока событий - запросы от пользовательских программ и прерывания от устройства. В простейшем

случае обслуживание обоих типов событий состоит в том, что у драйвера вызываются соответствующие функции: при формировании запроса на запись пользовательская программа (при посредстве диспетчера системных вызовов) зовет функцию write драйвера, а при обработке прерывания ядро системы (точнее, диспетчер прерывания) вызывает функцию interrupt того же самого драйвера. Защита от одновременного вызова этих функций возлагается на драйвер и часто состоит в том, что драйвер запрещает прерывания на время работы критических секций своей функции write. Такая архитектура используется драйверами символьных устройств в старых (так называемых "монолитных") ядрах Unix-систем.

При работе с блочными устройствами, а в более современных Unix-системах также с символьными устройствами STREAMS, используется более совершенная архитектура, когда запросы пользовательских программ ставятся в очередь к драйверу. Каждый запрос снабжается кодом запроса (чтение, запись, ioctl). В некоторых ОС, например в DEC RSX-11 и VMS, такая архитектура была единственно допустимой архитектурой для драйверов.

Многие драйверы вынуждены иметь дело с множественными потоками событий. Так, например, драйвер сетевого протокола IP в машине с несколькими сетевыми интерфейсами должен обслуживать потоки событий от драйверов всех этих интерфейсов, потоки событий от драйверов протоколов транспортного уровня, а также, возможно, потоки событий от приложений, работающих с сетью напрямую, через сокеты типа RAW.

Драйвер, допускающий одновременную работу с устройством нескольких пользовательских процессов, должен обеспечивать обслуживание нескольких потоков пользовательских запросов.

С формальной точки зрения обработчик событий удобнее всего описывать в виде конечного автомата. Конечный автомат описывается в виде набора допустимых состояний и набора допустимых переходов между ними, т.е. представляет собой ориентированный граф. Этот граф может описываться как матрицей инцидентности (таблицей состояний), так и картинкой, состоящей из коробочек (состояний) и стрелочек (переходов), например, диаграммой состояний UML. В

некоторых случаях применяют также диаграмму, в которой переходы ("активности") изображаются в виде коробочек, а состояния - в виде стрелочек, например диаграмма активностей UML (см. рис. 9.1). В действительности эти два типа диаграмм - два разных способа описания одного и того же.

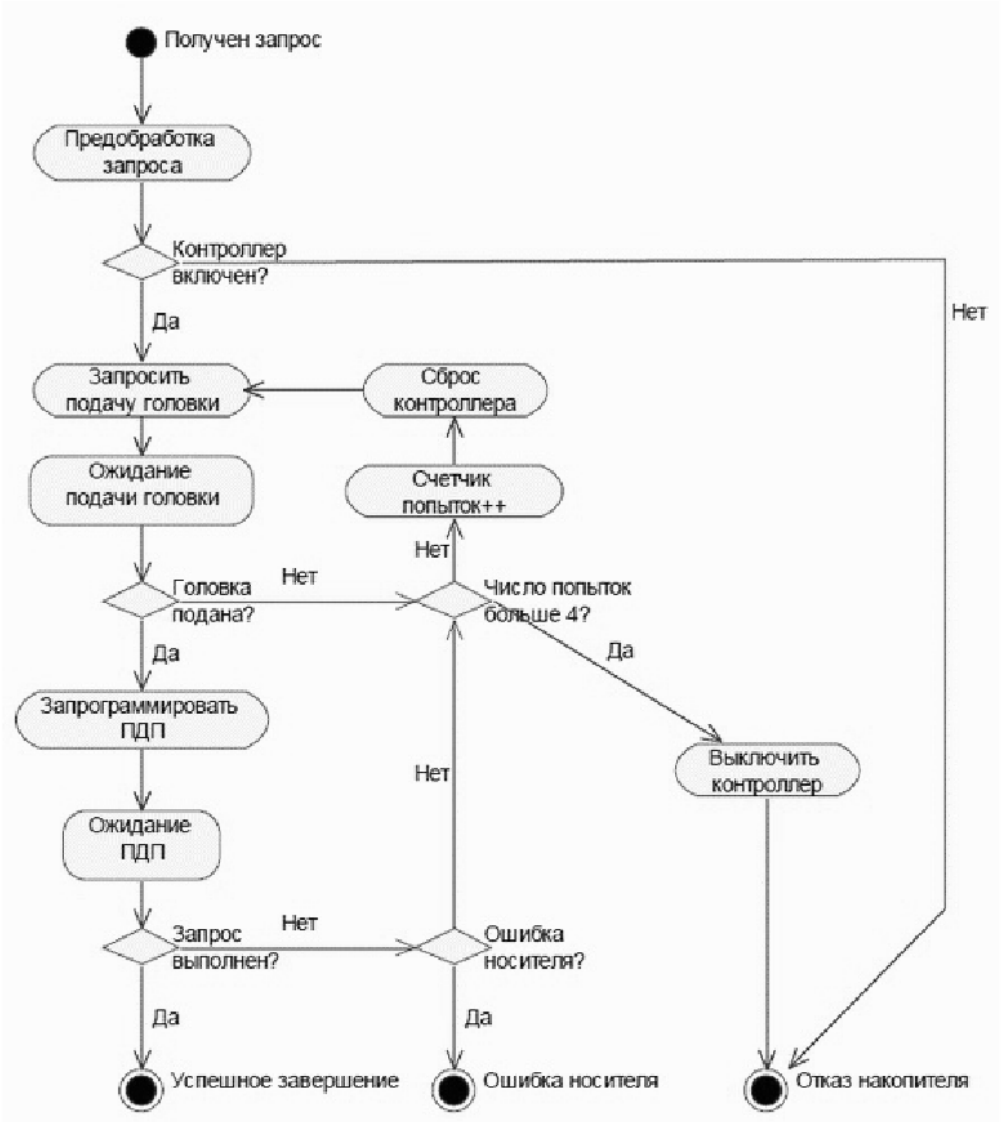


Рис. 9.1. Диаграмма активностей для обработки одного запроса драйвером привода для гибких магнитных дисков (для упрощения диаграммы опущено включение мотора и ожидание разгона диска до

рабочей скорости)

При реализации конечного автомата, состояние кодируется одной из переменных в блоке переменных состояния автомата, а переходы инициируются источниками событий. Если источник событий пытается инициировать недопустимый переход (послать событие, которое обработчик сейчас не может или не должен обрабатывать), возможны разные стратегии поведения. Соответствующее событие может просто игнорироваться (на графе состояний это изображают кольцевой стрелкой, выходящей из состояния и возвращающейся в то же самое состояние), порождать встречное сообщение об ошибке или ставиться в очередь. В некоторых случаях обработчик может блокировать нежелательные для него источники событий - например, в графическом интерфейсе контроллер диалогового окна может заблокировать кнопку OK, пока пользователь не введет в поля диалога все необходимые данные, а драйвер может заблокировать прерывания от устройства, установив соответствующую комбинацию бит в регистрах устройства.

Сравнение обработки событий и многопоточных приложений

С точки зрения нашего курса, интересно сравнить классические многопроцессные и многопоточные приложения с событийно-ориентированными.

В многопоточном приложении для каждой нити (потока) создается иллюзия последовательного исполнения. При переключении управления на другую нить система сохраняет контекст нити (в первую очередь стек и счетчик команд, но также и другие регистры процессора), а при возврате управления нашей нити - восстанавливает все эти регистры. Нить может хранить свое состояние в различных областях памяти - в регистрах, встеке,в `thread-specific data`. Если нити необходимо обработать внешнее (по отношению к ней) событие, она блокируется в ожидании этого события, исполняя блокирующийся системный вызов или библиотечную функцию. После выхода из блокировки ее исполнение продолжается с той же точки, в которой было остановлено.

В событийно-ориентированной архитектуре, для методов обработчика событий не создается иллюзии последовательного исполнения. Если методу обработчика надо перейти к обработке следующего события, он просто завершается. Когда следующее событие возникнет, система его позовет. Таким образом, обработчик событий не имеет собственного контекста. Он должен хранить все свое состояние в блоке переменных состояния.

Таким образом, главное отличие между традиционными многопоточными и событийно-ориентированными приложениями можно описать так: многопоточное приложение блокируется на примитивах взаимодействия с источниками событий с сохранением контекста нити. Событийно-ориентированное приложение блокируется на примитивах взаимодействия с источниками событий с уничтожением контекста нити. На практике, разумеется, контекст нити обычно не уничтожается, а переиспользуется для обработки других событий другими обработчиками.

Как правило, переиспользование контекста достигается за счет того, что нить, которая вызывает методы обработчиков событий, исполняет цикл менеджера или диспетчера событий. Менеджер событий опрашивает источники событий, при отсутствии событий - блокируется, а при появлении событий - диспетчеризует события обработчикам.

Из такого описания очевидно, что событийно-ориентированная архитектура гораздо дешевле при исполнении, чем классическая многопоточная при обработке того же потока событий. Ей требуется гораздо меньше ресурсов - памяти для хранения контекстов и процессорного времени для сохранения и восстановления этих контекстов - чем классической многопоточной архитектуре.

Из такого описания понятны также возможные подходы к построению гибридной архитектуры. Действительно, при всех своих достоинствах, событийно-ориентированная архитектура предъявляет довольно жесткие требования к обработчикам событий - они должны завершаться за небольшое время и нигде не блокироваться. Далеко не для всех задач легко выполнить эти требования. Обработка некоторых событий может требовать длительных вычислений. Другие события могут иметь такую природу, что их нельзя включить в цикл опроса диспетчера событий.

Например, если диспетчер событий опрашивает источники событий при помощи `select/poll`, мы не можем включить в этот цикл события, связанные с примитивами синхронизации POSIX ThreadsAPI. В третьем случае мы можем быть вынуждены в рамках обработки события запускать код, написанный другими программистами для других целей; этот код может ничего не знать про событийно-ориентированную архитектуру и про то, что ему нельзя блокироваться.

В этих случаях может оказаться целесообразной архитектура с несколькими менеджерами событий, исполняющимися в нескольких разных потоках.

Существует несколько вариантов такой архитектуры - асинхронная очередь сообщений, preforked processes, thread pool (пул нитей), рабочие нити (worker threads). В действительности, классификация вариантов этой архитектуры не является общепринятой и поэтому разные поставщики программного обеспечения часто называют одно и то же разными названиями, а разные архитектуры - одинаковыми терминами.

Так, например, популярный HTTP-сервер Apache 1.x имел следующую архитектуру. При запуске, основной процесс Apache привязывал сокет к порту 80 и, если это необходимо, создавал сокеты и привязывал их к другим обслуживаемым портам. Затем этот процесс несколько раз исполнял `fork(2)`, создавая свои копии. Все эти копии наследовали слушающие сокеты, и блокировались в вызове `accept(3SOCKET)` (в действительности, на `select(3C)` с ожиданием всех слушающих сокетов). При приходе запроса на соединение, у одного из процессов `accept` разблокировался и этот процесс начинал обработку этого запроса. Разумеется, обработка запроса включает в себя многочисленные блокировки на вызовах `read(2)` и `write(2)` ; кроме того, обработка может включать в себя загрузку модулей и cgi-скриптов, которые исполняют практически произвольный код и, вообщеговоря, могут блокироваться на чем угодно, чаще всего - на обращениях к базе данных. Если в это время приходит еще один запрос, его должен подхватить другой процесс.

Когда процесс Apache завершает обработку запроса, он не завершается, а возвращается к вызову `accept` и блокируется в ожидании следующего соединения. Таким образом, сервер не тратит время и ресурсы на запуск

процесса при поступлении каждого нового запроса на соединение, но, тем не менее, каждое соединение все равно обслуживается отдельным процессом. Именно эта архитектура и называется preforked processes, или, если вместо процессов мы используем нити, пулом нитей (thread pool). Видно, что это частный случай событийно-ориентированной архитектуры, при которой обрабатываемым событием считается приход нового запроса на соединение.

Количество процессов, запускаемых Apache, подбирается эмпирически исходя из производительности сервера, потока запросов, скорости каналов, по которым приходят эти запросы, и среднего времени исполнения скриптов на страницах веб-сайта.

Несмотря на простоту этой архитектуры, она обладает очевидным недостатком. Если все процессы сервера заблокированы на том или ином системном вызове, это означает, что сервер имеет ресурсы (во всяком случае, процессорное время) для обработки еще одного соединения - но для этого ему надо запускать еще один процесс!

Чтобы обойти эту проблему, нам следует считать отдельным событиемне приход нового запроса на соединение, а готовность каждого из соединений к приему и передаче данных. На предыдущей лекции мы изучали средства, которые специально предназначены для диспетчеризации таких событий - `select(3C)`, `poll(2)`, порты Solaris. Можно также считать событием не готовность соединений к приему и передаче данных, а завершение предыдущего запроса или группы запросов к этому соединению. В таком случае необходимо использовать асинхронный ввод/вывод, а для диспетчеризации событий можно использовать `aio_suspend(3AIO)`, `aio_wait(3AIO )`, `sigwait(2)` (в сочетании с использованием сигналов для оповещения о завершении запроса) или опять-таки порты Solaris.

Обсуждаемую архитектуру невозможно реализовать при использовании preforked процессов. Действительно, чтобы один процесс мог обслуживать несколько соединений, он либо должен сделать `accept(3SOCKET)` на все эти соединения, либо должен быть запущен уже после того, как родитель сделает этот `accept`. Оба варианта противоречат самой идее preforked processes. Поэтому

наиболее естественным вариантом для реализации предлагаемой архитектуры является многопоточное приложение.

Реализация кэширующего прокси.

Рассмотрим приведенные выше общие рассуждения применительно к конкретной задаче разработки кэширующего прокси для протокола HTTP 1.0. В рамках данной задачи мы будем делать упрощенный вариант прокси, хранящий кэш в оперативной памяти. (дальнейшее обсуждание предполагает, что слушатель знает, как устроен протокол HTTP, что такое прокси, а также знаком с основами программирования сетевых приложений с использованием сокетов TCP/IP. Если слушатель с этим не знаком, на этом занятие можно закончить). Разумеется, при перезапуске прокси такой кэш будет теряться, что сильно ограничивает практическое применение нашей программы. но это избавит нас от многих проблем, например связанных с обеспечением согласования содержимого кэша при аварийных остановках.

Ограничение версии протокола HTTP 1.0 позволяет нам отказаться от персистентных соединений, что также упростит реализацию.

Прежде чем пытаться представить себе приложение в целом, давайте сначала попытаемся представить себе последовательность событий при обработке одиночного запроса. Сначала мы опишем этот процесс на естественном языке, а затем нарисуем его в виде конечного автомата. В действительности, большинство сетевых протоколов явно или неявно предполагают реализацию клиента и сервера в виде конечных автоматов, так что такое описание будет для нашей задачи понятным и естественным. Затем мы обсудим вопрос о том, как организовать одновременное исполнение множества таких автоматов в рамках событийно-ориентированной архитектуры. В зависимости от того, как мы будем организовывать это исполнение, детали реализации механизмов передачи событий могут различаться.

Первым делом нам следует решить вопрос, будем ли мы обрабатывать соединения клиента с прокси и прокси с сервером одним и тем же обработчиком, или двумя разными. На данном этапе мы не знаем, какое из решений будет правильным. Поэтому мы примем решение волевым актом или, проще говоря, наугад. Если при попытке довести это

решение до реализации мы столкнемся с проблемами, нам надо будет вернуться назад и пересмотреть его. Будем надеяться, что мы сможем обнаружить проблемы еще при обсуждении проекта, т.е. , до того, как мы напишем и отладим хотя бы одну строчку кода.

Итак, волевым актом мы принимаем решение, что соединения клиента с прокси и прокси с сервером будут обрабатываться двумя разными источниками событий.

Рассмотрим возможные последовательности событий с точки зрения клиентского соединения.

Работа обработчика событий клиентскогосоединения начинается с прихода запроса от клиента. Скорее всего, мы обнаруживаем это по тому факту, что слушающий сокет нашего прокси "готов к чтению" с точки зрения `select/poll`.

Мы принимаем запрос на соединение при помощи accept и переходим в состояние ожидания запроса. Обычные браузеры, такие, как Internet Explorer или Mozilla Firefox, присылают запрос сразу после установления соединения, но при тестировании мы можем захотеть присоединиться к нашему прокси при помощи команды `telnet(1)` и набрать команду вручную. При этом, разумеется, первая строка запроса придет через весьма значительное(с точки зрения компьютера) время после установления соединения. Тот же эффект может возникать, если клиент присоединяется к нашему прокси по медленному каналу, или клиенту плохо по каким-то внутренним причинам (например он работает на Windowscистеме с малым объемом памяти).

Так или иначе, рано или поздно мы должны получить заголовок HTTP-запроса. Первая строка запроса обязана иметь формат `[команда] [url] [версия HTTP]` и заканчиваться символами " `\r\n` ". Для упрощения отладки нам, наверное, не следует слишком удивляться, если вместо " `\r\n` " клиент пришлет нам только " `\n` ". При тестировании прокси нам вряд ли удастся добиться, чтобы первая строка запроса пришла к нам по частям - типичная первая строка запроса меньше MTU распространенных протоколов канального уровня и меньше минимально допустимого MTU IPv4. К тому же, если набирать строку в `telnet`, надо иметь в виду, что `telnet` по умолчанию буферизует

ввод по строкам. Тем не менее, мы обязаны быть готовы к тому, что протокол транспортного уровня по неведомым нам причинам разрежет первую строку запроса на части и мы вынуждены будем делать `read(2)` несколько раз, чтобы собрать ее.

По первой строке запроса мы уже можем решить, что нам делать с этим запросом. Во первых, мы можем увидеть версию HTTP/1.1, которую по техзаданию мы поддерживать не обязаны. На это мы обязаны ответить кодом ошибки 505 (HTTP Version not supported). Умный браузер послеэтого обязанповторить запрос с версией 1.0. Некоторыебраузеры, например lynx, могут прислать запрос с версией 0.9 - на самом деле, функций, поддерживаемых lynx, вполне достаточно для общения с ним как с браузером 1.0, но на такой запрос мы тоже имеем право ответить ошибкой 505. После этого мы с чистой совестью можем закрыть соединение и завершиться. Может быть нам даже не следует считывать остаток запроса.

Во вторых, по первой строке мы определяем метод запроса. Для упрощения прокси нам следует реализовать только методы `GET` и, наверное, `HEAD` (многие браузеры шлют запрос `HEAD` чтобы определить, изменилась ли страница и следует ли ее скачивать целиком, или можно взять копию из локального кэша браузера). В действительности, как мы увидим далее, ничто не мешает нам реализовать также методы `POST` и `PUT`. Метод `CONNECT`, напротив, нам, скорее всего реализовать вообще не следует - без управления доступом этот метод может использоваться для несанкционированного подключения к вашей сети в обход файрволла, а расширять наш прокси средствами фильтрации и управления доступом мы не успеем за время, отведенное на практикум.

Итак, если мы обнаруживаем метод, который наш прокси реализует, мы должны продолжать обработку. Если же мы не поддерживаем такой метод, нам следует вернуть ошибку 405 и закрыть соединение.

Методы `POST` и `PUT` не следует кэшировать ни при каких обстоятельствах. Для них следует открывать сквозное соединение и транслировать все проходящие через нас в обоих направлениях данные, подобно тому, как это надо сделать в упражнении 25.

С методами `GET` и `HEAD` не все так просто - в зависимости от того, что нам ответит сервер, может выясниться, что кэшировать страницу нельзя. Но мы будем писать очень простой прокси, который будет кэшировать все подряд. Количество сайтов, более или менее нормально работающих через такой прокси, удивительно велико.

Итак, при получении запроса `GET` и `HEAD` нам следует проверить, есть ли такая страница в кэше. На этот вопрос может быть три ответа:

1. Страницы нет в кэше
2. Страница есть в кэше и она закачана в кэш полностью.
3. Страница есть в кэше, но она закачана не полностью.

При проверке того, есть ли страница в кэше, по хорошему следует использовать в качестве ключа не только URL, но также куки (`cookie`) и параметры авторизации - в зависимости от пользователя и кук страница может выглядеть по разному. Но мы делаем очень простой прокси, который не поддерживает ни авторизацию, ни куки, так что мы будем использовать в качестве ключа один только URL.

При разборе URL мы можем также обнаружить, что используется протокол, отличный от http. На это нам тоже следует ответить ошибкой.

Итак, большинство ошибок можно обнаружить уже по первой строке запроса.

Но если ошибки нет, то у нас получаются три разные ветви исполнения, которые соответствуют трем разным выходам из состояния "ожидание запроса":

1. Страницы нет в кэше

   В этом случае надо выделить буфер под весь HTTP-запрос вместе со всеми его дополнительными полями, и послать сообщение системе, что нам нужен обработчик запроса к серверу.

   После этого, очевидно, надо дождаться какого-то ответа на запрос - ведь может выясниться, что такого сервера нет в сети, что он не отвечает на соединения или что он не может ничего хорошего

ответить на наш запрос. Если нам вообще не удалось получить ответ от сервера, нам следует отдать клиенту заранее заготовленную страницу, вроде тех, которые отдают настоящие прокси, и завершить соединение. Если же ответ удалось получить, но он отличается от `200`, его следуетпередать клиенту и тоже завершить соединение.

Если же получен ответ `200`, обработчик запроса к серверу должен создать новую запись кэша, и сообщить нам об этом. Тогда наше соединение переходит в состояние варианта 3 - запись есть в кэше, но закачана туда не полностью.

2. Страница есть в кэше и она закачана в кэш полностью

   Это самый простой вариант. Настоящий прокси, конечно, должен озаботиться еще одной вещью - он должен послать на сервер запрос `HEAD`, чтобы выяснить, не изменилась ли страница, и если изменилась, то задача сводится к варианту "страницы нет в кэше". Но наш игрушечный прокси может сразу отдавать страницу клиенту, ориентируясь только на длину страницы и на готовность клиента получать данные.

3. Страница есть в кэше, но она закачана не полностью

   В этом случае тоже все просто. У нас есть два источника событий - готовность клиента получать данные и сообщение обработчика событий от сервера, что он получил еще данные. Сообщения от обработчика событий сервера могут быть двух типов - "получены еще данные и страница еще не докачана" и "получены еще данные и страница докачана полностью". В рамках протокола HTTP 1.0 мы не можем отличить второй вариант от аварийного завершения работы с сервером (то и другое выглядит как разрыв соединения).

   Интересный вариант возникает, если мы еще передали клиенту не всю страницу, но клиент завершил соединение (это происходит довольно часто, например когда клиент нажимает на ссылку на странице, не дожидаясь докачки всей страницы).

   Наверное, в этом случае нам следует оповестить серверный обработчик событий, что страница нам не нужна… Хотя… есть

возможность, что эту же страницу качают другие клиенты, поэтому закрытие соединения одним клиентом, строго говоря, не повод закрывать соединение с сервером. То есть серверный обработчик должен знать, сколько у него клиентов, то есть должны быть какие-то еще сообщения между обработчиками. Чем больше об этом думаешь, тем сложнее все становится. Наверное, нам следует пока что остановиться на варианте, когда серверный обработчик события в любом случае докачивает страницу до конца.

Итак, на первый взгляд мы рассмотрели все варианты развития событий на стороне клиентского соединенияи можем переходить к разработке вариантов развития событий на стороне серверного соединения.

Для серверного соединенияработа начинается, когда кто-то из клиентов обнаружил, что нужной ему записи нет в кэше. При этом клиент должен предоставить полный текст HTTP запроса со всеми полями заголовка (запросы `PUT` и `POST`, мы, как договорились, не рассматриваем). По хорошему, сервер должен добавить к этим полям свой идентификатор, но мы пишем очень простой прокси, который не будет этого делать и просто передаст этот запрос серверу без всяких изменений.

Если соединение с сервером установить не удастся или сервер ответит нам что-то плохое, мы должны оповестить об этом клиента. Если сервер ответит нам `200`, мы должны тоже оповестить об этом клиента, выделить буфер под запись кэша и начать скачивать страницу.

Однако тут-то нас ждет подводный камень. Некоторые страницы честно говорят свой размер в заголовке HTTP-запроса. Но HTTP 1.0 допускает страницы без размера в заголовке. Размер такой страницы определяется косвенно по разрыву соединения сервером. Но мы, разумеется, не знаем, когда сервер его разорвет, поэтому, наверное, нам придется переразмещать буфер при помощи *`realloc`*`(3C)`. Но ведь клиент читает данные из этого буфера - то есть нам надо предусмотреть сообщение клиенту -всем клиентам! - что местоположение буфера изменилось. Это все немного усложняет для клиента - впрочем, не фатально. Наверное, клиенту вместо указателя на текущую позицию вбуфере надо было бы хранить смещение текущей позиции. Либо нам

надо реализовать запись кэша в виде списка буферов, а не в виде одного линейного буфера. Ну и в многопоточной версии прокси надо бы озаботиться защитой этого места чем-то вроде мутексов… Но пока что все это выглядит разрешимым, хотя, конечно, придется написать некоторое количество дополнительного кода…

К сожалению, в Интернете немало страниц, на которых лежат очень большие объекты.

Например, на сайтах, на которых раздают дистрибутивы свободно распространяемых Unix-систем, часто можно найти файлы с расширением `.iso` размером от 600 мегабайт до четырех гигабайт, а по мере распространения HD-DVD и BluRay, наверное, можно будет обнаружить файлы и большего размера.. На 32-разрядной машине вам под такое не дадут памяти никогда. Да и на 64-разрядных машинах настройки ресурсных квот могут вам не позволить получить такой объем памяти.

Наверное, если мы обнаружим такой файл, нам следует переводить соединение из кэшируемого в сквозноеи оповещать об этом клиента. Но что делать, если к моменту осознания размеров файла с этой записью кэша работало несколько клиентских соединений???

Задав себе этот вопрос мы, наверное, вынуждены будем закончить лекцию, потому что изложить варианты ответа на этот вопрос и найти среди них удовлетворительный мы уже не успеем.

Первые три студента в группе, которые будут сдавать прокси, будут тестироваться на сайтах с маленькими страницами, которые гарантированно влезают в оперативную память.

Еще несколько замечаний по прокси

Важной проблемой при реализации однопоточного прокси являются запросы к DNS. Стандартизованного API для асинхронных обращений к DNS не существует, да и интеграция нестандартных API с событийно-ориентированной архитектурой на `select/poll` достаточно сложна, поэтому с этим недостатком придется смириться.

Для асинхронного установления соединения, напротив, существует стандартный API, так называемые неблокирующиеся сокеты. Сокет объявляется неблокирующимся при помощи `fcntl(2)` с флагом `O_NONBLOCK`. `Connect(3SOCKET)` с таким сокетом возвращает ошибку `EINPROGRESS`. Обнаружить факт установления соединения можно при помощи `select(3C)` или `poll(2)` проверкой готовности сокета для записи. Неблокирующийся `connect(3SOCKET)` целесообразно использовать и в однопоточных, и в многопоточных версиях прокси.

После установления соединения флаг `O_NONBLOCK` следует снять, во всяком случае для отладки. Иначе есть риск, что некорректная проверка этого сокета в `select(3C)/poll(2)` выродится в холостой цикл.

Для стресс-тестирования вашего прокси можно использовать `wget(1)` с ключом -- `mirror`.

Архитектуры многопоточных приложений

В заключение нашего курса рассмотрим реально используемые архитектуры многопоточных приложений.

Многопроцессные приложения с автономными процессами

Это самый простой тип многопроцессных приложений. Для каждой пользовательской сессии или даже для каждого запроса создается свой процесс. Он обрабатывает запрос или несколько запросов и завершается.

Иногда такие процессы "взаимодействуют" при записи сообщений в лог-файл. Для разрешения возникающих при этом коллизий в Unix-системах предусмотрен флаг `O_APPEND` при открытии файла. При записи в файлы, открытые с этим флагом, указатель записи всегда перемещается на конец файла. Таким образом, записи в логе, выполняемые разными процессами, никогда несмешиваются. В более современныхUnix-системах для ведения логов предоставляется специальный сервис `syslog(3C)`.

Преимущества:

1. Простота разработки. Фактически, мы запускаем много копий однопоточного приложения и они работают независимо друг от друга. Можно не использовать никаких специфически многопоточных API и средств межпроцессного взаимодействия.
2. Высокая надежность. Аварийное завершение любого из процессов никак не затрагивает остальные процессы.
3. Хорошая переносимость. Приложение будет работать налюбой многозадачной ОС
4. Высокая безопасность. Разные процессы приложения могут запускаться от имени разных пользователей. Таким образом можно реализовать принцип минимальных привилегий, когда каждый из процессов имеет лишь те права, которые необходимы ему для работы. Даже если в каком-то из процессов будет обнаружена ошибка, допускающая удаленное исполнение кода,

взломщик сможет получить лишь уровень доступа, с которым исполнялся этот процесс.

Недостатки:

1. Далеко не все прикладные задачи можно предоставлять таким образом. Например, эта архитектура годится для сервера, занимающегося раздачей статических HTMLстраниц, но совсем непригодна для сервера баз данных и многих серверов приложений.
2. Создание и уничтожение процессов – дорогая операция, поэтому для многих задач такая архитектура неоптимальна.

В Unix-системах предпринимается целый комплекс мер для того, чтобы сделать создание процесса и запуск новой программы в процессе как можно более дешевыми операциями. Однако нужно понимать, что создание нити в рамках существующего процесса всегда будет дешевле, чем создание нового процесса.

Примеры: apache 1.x (сервер HTTP)

Многопроцессные приложения, взаимодействующие через сокеты, трубы и очереди сообщений System V IPC

Перечисленные средства IPC (Interprocess communication) относятся к так называемым средствам гармонического межпроцессного взаимодействия. Онипозволяют организовать взаимодействие процессов и потоков без использования разделяемой памяти. Теоретики программирования очень любят эту архитектуру, потому что она практически исключает многие варианты ошибок соревнования.

Преимущества:

1. Относительная простота разработки.
2. Высокая надежность. Аварийное завершение одного из процессов приводит к закрытию трубы или сокета, а в случае очередей сообщений – к тому, что сообщения перестают поступать в очередь или извлекаться из нее. Остальные процессы приложения

легко могут обнаружить эту ошибку и восстановиться после нее, возможно (но не обязательно) просто перезапустив отказавший процесс.

3. Многие такие приложения (особенно основанные на использовании сокетов) легко переделываются для исполненияв распределенной среде, когда разные компоненты приложения исполняются на разных машинах.
4. Хорошая переносимость. Приложение будет работать на большинстве многозадачных ОС, в том числе на старых Unix-системах.
5. Высокая безопасность. Разные процессы приложения могут запускаться от имени разных пользователей. Таким образом можно реализовать принцип минимальных привилегий, когда каждый из процессов имеет лишь те права, которые необходимы ему для работы.

Даже если в каком-то из процессов будет обнаружена ошибка, допускающая удаленное исполнение кода, взломщик сможет получить лишь уровень доступа, с которым исполнялся этот процесс.

Недостатки:

1. Не для всех прикладных задач такую архитектуру легко разработать и реализовать.
2. Все перечисленные типы средств IPC предполагают последовательную передачу данных. Если необходим произвольный доступ к разделяемым данным, такая архитектура неудобна.
3. Передача данных через трубу, сокет и очередь сообщений требует исполнения системных вызовов и двойного копирования данных – сначала из адресного пространства исходного процесса в адресное пространство ядра, затем из адресного пространства ядра в память целевого процесса. Это дорогие операции. При передаче больших объемов данных это может превратиться в серьезную проблему.
4. В большинстве систем действуют ограничения на общее количество труб, сокетов и средств IPC. Так, в Solaris по умолчанию допускается не более 1024 открытых труб, сокетов и файлов на процесс (это обусловлено ограничениями системного

вызова select). Архитектурное ограничение Solaris – 65536 труб, сокетов и файлов на процесс.

Ограничение на общее количество сокетов TCP/IP – не более 65536 на сетевой интерфейс (обусловлено форматом заголовков TCP). Очереди сообщений System V IPC размещаются вадресном пространствеядра, поэтому действуют жесткиеограничения на количество очередей в системе и на объем и количество одновременно находящихся в очередях сообщений.

5. Создание и уничтожение процесса, а также переключение между процессами – дорогие операции. Не во всех случаях такая архитектура оптимальна.

Многопроцессные приложения, взаимодействующие через разделяемую память

В качестве разделяемой памяти может использоваться разделяемая память System V IPC и отображение файлов на память. Для синхронизации доступа можно использовать семафоры System V IPC, мутексы и семафоры POSIX, при отображении файлов на память – захват участков файла.

Преимущества:

1. Эффективный произвольный доступ к разделяемым данным. Такая архитектура пригодна для реализации серверов баз данных.
2. Высокая переносимость. Может быть перенесено налюбую операционную систему, поддерживающую или эмулирующую System V IPC.
3. Относительно высокая безопасность. Разные процессыприложениямогут запускаться от имени разных пользователей. Таким образом можно реализовать принцип минимальных привилегий, когда каждый из процессов имеет лишь те права, которые необходимы ему для работы. Однако разделение уровней доступа не такое жесткое, как в ранее рассмотренных архитектурах.

Недостатки:

1. Относительная сложность разработки. Ошибки при синхронизации доступа – так называемые ошибки соревнования – очень сложно обнаруживать при тестировании.

   Это может привести к повышению общей стоимости разработки в 3–5 раз по сравнению с однопоточными или более простыми многозадачными архитектурами.

2. Низкая надежность. Аварийное завершение любого из процессов приложения может оставить (и часто оставляет) разделяемую память в несогласованном состоянии.

   Это часто приводит к аварийному завершению остальных задач приложения. Некоторые приложения, например Lotus Domino, специально убивают всесерверные процессы при аварийном завершении любого из них.

3. Создание и уничтожение процесса и переключение между ними – дорогие операции.

   Поэтому данная архитектура оптимальна не для всех приложений.

4. При определенных обстоятельствах, использование разделяемой памяти может приводить к эскалации привилегий. Если в одном из процессов будет найдена ошибка, приводящая к удаленному исполнению кода, с высокой вероятностью взломщик сможет ее использовать для удаленного исполнения кода в других процессах приложения.

   То есть, в худшем случае, взломщик может получить уровень доступа, соответствующий наивысшему из уровней доступа процессов приложения.

5. Приложения, использующие разделяемую память, должны исполняться на одном физическом компьютереили, во всяком случае, на машинах, имеющих разделяемое ОЗУ. В действительности, это ограничение можно обойти, например используя отображенные на память разделяемые файлы, но это

приводит к значительным накладным расходам

Фактически, данная архитектура сочетает недостатки многопроцессных и собственно многопоточных приложений. Тем не менее, ряд популярных приложений, разработанных в 80е и начале 90х, до того, как в Unix были стандартизованы многопоточные API, используют эту архитектуру. Это многие серверы баз данных, как коммерческие (Oracle, DB2, Lotus Domino), такисвободно распространяемые,современные версии Sendmail инекоторые другие почтовые серверы.

Собственно многопоточные приложения

Потоки или нити приложения исполняются в пределах одного процесса. Все адресное пространство процесса разделяется между потоками. На первый взгляд кажется, что это позволяет организовать взаимодействие между потоками вообще без каких-либо специальных API. В действительности, это не так – если несколько потоков работает с разделяемой структурой данных или системным ресурсом, и хотя бы один из потоков модифицирует эту структуру, то в некоторые моменты времени данные будут несогласованными.

Поэтому потоки должны использовать специальные средства для организации взаимодействия. Наиболее важные средства – это примитивы взаимоисключения (мутексы и блокировки чтения-записи). Используя эти примитивы, программист может добиться того, чтобы ни один поток не обращался к разделяемым ресурсам, пока они находятся в несогласованном состоянии (это и называется взаимоисключением). POSIX thread library предоставляет и другие примитивы (например, условные переменные), позволяющие организовать более сложные, чем взаимоисключение, схемы взаимодействия между нитями.

Преимущества:

1. Высокая производительность. На большинстве Unix-систем, создание нити требует в десятки раз меньше процессорного времени, чем создание процесса.
2. Эффективный произвольный доступ к разделяемым данным. В частности, такая архитектура пригодна для создания серверов баз

данных.

3. Высокая переносимость илегкость переноса ПОиз-под других ОС, реализующих многопоточность.

Недостатки:

1. Высокая вероятность опасных ошибок. Все данные процесса разделяются между всеми нитями. Иными словами, любая структура данных может оказаться разделяемой, даже если разработчик программы не планировал этого (для сравнения, в многопроцессных приложениях, использующих разделяемую память System V IPC, разделяются только те структуры, которые размещены в сегменте разделяемой памяти. Обычные переменные и размещаемые обычным образом динамические структуры данных свои укаждого изпроцессов). Ошибки придоступекразделяемым данным – ошибки соревнования – очень сложно обнаруживать при тестировании.
2. Высокая стоимость разработки и отладки приложений, обусловленная п. 1.
3. Низкая надежность. Разрушение структур данных, например в результате переполнения буфера или ошибок работы с указателями, затрагивает все нити процесса и обычно приводит к аварийному завершению всего процесса. Другие фатальные ошибки, например, деление на ноль в одной из нитей, также обычно приводят к аварийной остановке всех нитей процесса.
4. Низкая безопасность. Все нити приложения исполняются в одном процессе, то есть от имени одного и того же пользователя и с одними и теми же правами доступа. Невозможно реализовать принцип минимума необходимых привилегий, процесс должен исполняться от имени пользователя, который может исполнять все операции, необходимые всем нитям приложения.
5. Создание нити – все-таки довольно дорогая операция. Для каждой нити в обязательном порядке выделяется свой стек, который по умолчанию занимает 1 мегабайт ОЗУ на 32битных архитектурах и 2 мегабайта на 64-битных архитектурах, и некоторые другие ресурсы. Поэтому данная архитектура оптимальна не для всех приложений.
6. Невозможность исполнять приложение на многомашинном вычислительном комплексе. Упоминавшиеся в предыдущем

разделе приемы, такие, как отображение на память разделяемых файлов, для многопоточной программы не применимы.

В целом можно сказать, что многопоточные приложения имеют почти те же преимущества и недостатки, что и многопроцессные приложения, использующие разделяемую память.

Однако стоимость исполнения многопоточного приложения ниже, а разработка такого приложения в некоторых отношениях проще, чем приложения, основанного наразделяемой памяти. Поэтому в последние годы многопоточные приложения становятся все более и более популярны.

Событийно-ориентированные архитектуры

В событийно-ориентированной архитектуре мы отказываемся от взгляда на исполнение программы как на последовательный процесс. Вместо этого мыреализуем программу ввиде набора функций или методов-обработчиков, которые вызываются в ответ на возникновение событий. События могут быть как внешними по отношению к нашей программе (например, прибытие порции данных из сетевого соединения, сработка таймера или нажатие клавиши пользователем), так и внутренними. Внутренние события используются для взаимодействия между разными обработчиками.

Как правило, событийно-ориентированная архитектура предполагает наличие специального модуля, называемогодиспетчером илименеджером событий. Этот модуль темили иным образом собирает информацию о возникших событиях, организует сообщения о событиях и обработчики в очереди в соответствии с теми или иными правилами и вызывает обработчики.

Главная сложность при разработке событийно-ориентированного приложения – это гарантировать, что все обработчики событий будут завершаться достаточно быстро.

Как правило это означает, что обработчики не должны вызывать блокирующихся системных вызовов. Если удастся соблюсти это требование, то событийно-ориентированная архитектура позволяет

получить многие преимущества многозадачности и многопоточности без использования собственно многопоточности. В этом смысле можно даже сказать, что событийно-ориентированная архитектура – это альтернатива многозадачным и многопоточным приложениям.

Событийно-ориентированные архитектуры применяются во множестве различных случаев. Таким образом реализуются программы с графическим пользовательским интерфейсом, сетевые серверы и некоторые клиентские сетевые программы, приложения реального времени, игры и др. Ядра большинства современных операционных систем, таких, как Windows, Linux и BSD Unix, также имеют событийно-ориентированную архитектуру.

Преимущества

1. Высокая производительность. Грамотно разработанное событийно-ориентированное приложение может одновременно обрабатывать множество событий в рамках одного потока и одного процесса.

   Некоторые событийно-ориентированные приложения обрабатывают сотни итысячи сетевых соединений в одном потоке.

Недостатки:

1. Не для всех приложений эта архитектура подходит. Так, если обработка события требует длительных вычислений или ее невозможно реализовать без использования блокирующихся системных вызовов, это может затормозить обработку других событий.
2. Разработка событийно-ориентированного приложения требует высокой квалификации разработчиков. На практике это может привести, и обычно приводит, к высокой стоимости разработки.
3. Код, рассчитанный на другую архитектуру (например, использующий блокирующиеся системные вызовы), невозможно переиспользовать в событийно-ориентированном приложении. Напротив, многие из приемов кодирования, необходимых в событийно-ориентированных приложениях, бесполезны и даже

вредны в других архитектурах.

4. Низкая надежность. Фатальная ошибка при обработке любого события приводит к аварийному завершению всего процесса.
5. Низкая безопасность. Поскольку все события обрабатываются одним процессом, то все обработчики работают от имени одного и того же пользователя. Невозможно реализовать принцип минимально необходимых привилегий.

Гибридные архитектуры

Гибридные архитектуры отличаются большим разнообразием. Они позволяют сочетать преимущества, характерные для различных типов простых архитектур. Так, разработчик интерактивного приложения может реализовать пользовательский интерфейс в рамках событийно-ориентированной архитектуры, а для сложных вычислений (например, для переразбиения текста на страницы) запускать фоновые нити.

Однако важно понимать, что очень часто гибридные архитектуры сочетают не только преимущества, но и недостатки используемых базовых архитектур. Поэтому создание приложения с гибридной архитектурой предъявляет высокие требования к квалификации архитектора приложения и большинства разработчиков.

В конце лекции 9 мы рассматривали архитектуру "рабочих нитей" (worker threads), сочетающую многопоточность с событийной ориентацией. Ряд приложений, в том числе Apache 2.0, используют такую архитектуру.

Архитектуры вычислительных программ

Ранее мы рассматривали преимущества и недостатки многопроцессных и многопоточных архитектур с точки зрения приложений, занимающихся обработкой внешних событий – серверных и интерактивных приложений. Однако большой интерес представляют также вычислительные приложения, предназначенные для научных и инженерных расчетов.

Такие приложения не занимаются обработкой внешних событий и в

ряде отношений работают в тепличных условиях (по сравнению, скажем, с серверными приложениями, подключенными к Интернет). Так, разработчика вычислительной программы, как правило, не интересуют вопросы безопасности: его задача работает с минимальным уровнем привилегий на выделенном компьютере и не взаимодействует с внешним миром, поэтому заниматься ее взломом обычно и неинтересно, и практически невозможно.

Однако и для вычислительных задач не существует единой оптимальной архитектуры, приемлемой для всех случаев.

Решающим вопросом при выборе архитектуры является вопрос о том, насколько хорошо распараллеливается алгоритм, какой объем разделяемых данных должны иметь процессы или нити и как часто им нужно синхронизовать эти данные.

Важным параметром задачи является также предсказуемость времени исполнения ее частей. Действительно, если время исполнения нитей алгоритма строго одинаково, мы легко можем распределить эти нити между процессорами или между машинами вычислительного кластера.

Если же время исполнения нитей различается и при этом трудно предсказуемо, у нас может возникнуть желание переносить нити или задачи между процессорами во время исполнения.

Такая ситуация возникает при расчете трехмерных изображений методом трассировки лучей с реалистичным моделированием рассеивания света. Перенос процессов сопряжен с большими сложностями и хорошие алгоритмы балансировки загрузки не известны (даже при известных временах исполнения процессов задача их распределения по процессорам является NP-полной, так называемой "задачей о рюкзаке"), поэтому разработчики параллельных задач все-таки стараются обеспечить равномерное распределение расчетов между процессами или нитями. В случаях, когда это не удается, обычно создается большое количество процессов – значительно большее, чем количество процессоров – и затем эти процессы распределяются между процессорами случайным образом.

Примеры очень хорошо распараллеливающихся задач с малым объемом разделяемых данных – взлом шифра методом "грубой силы" (например,

перебором пространства ключей) или вычисление интеграла методом Монте-Карло. Примеры вовсе не распараллеливаемых задач – вычисление ряда, определяемого индуктивной формулой `a(n)=f(a(n-1))` или поиск минимума многомерной функции методом градиентного спуска.

Два основных подхода к реализации параллельных вычислительных программ – это параллельные программы с разделяемой памятью и программы, обменивающиеся сообщениями.

Программы с разделяемой памятью удобны, когда нити алгоритма должны часто обмениваться большими объемами данных и/или осуществлять произвольный доступ к разделяемым данным большого объема. Программы, обменивающиеся сообщениями, удобны, когда объем разделяемых данных невелик, а эти данные изменяются редко и в предсказуемых местах.

Крайний случай программ, обменивающихся сообщениями –это уже упоминавшийся взлом шифра, когда каждая программа при запуске получает свой участок работы (дешифруемое сообщение, критерий успешности расшифровки и диапазон пространства ключей), выполняет свою часть работы независимо отдругих и сообщает о результате.

Наиболее распространенная технология параллельных вычислений с разделяемой памятью – OpenMP. Приложения OpenMP разрабатываются на специальных диалектах языков C и Fortran. Эти диалекты отличаются от базовых языков наличием специальных директив (в C это `#pragma omp`), описывающих, какие участки кода следует исполнять параллельно и какие переменные должны разделяться между нитями. Кроме директив компилятора OpenMP предоставляет также API для взаимодействия нитей. Некоторые компиляторы, например Sun Studio 11, могут автоматически распараллеливать программы, написанные на обычном C и Fortran. OpenMP использует `fork-join` модель многопоточности, когда программа в начале блока разветвляетсяна нескольконитей, а вконце блока собирает результаты исполнения этих нитей.

Таким образом, параллелизм программы имеет блочную структуру, аналогичную структуре кода в C и Java. Разумеется, это резко сужает

количество возможных программ, но в то же время это значительно упрощает разработку и отладку. Поэтому считается, что OpenMP предоставляет более высокоуровневую модель многопоточности, чем POSIX Threads.

OpenMP позволяет управлять количеством нитей программы во время запуска при помощи переменной среды `OMP_NUM_THREADS`. Сама программа также может управлять количеством нитей через специальный API. Как правило, рекомендуется, чтобы количество нитей программы на машине с `N` процессорами было равно `N` или `N-1`. По умолчанию программа OpenMP запускается с количеством нитей, которое равно количеству доступных процессоров. Если несколько пользователей запускают свои задачи на одной машине, им следует договориться об используемом количестве процессоров. Solaris предоставляет средства для принудительного управления количеством процессоров, доступных пользователю, такие, как зоны, контейнеры и проекты.

Программы OpenMP исполняются как несколько потоков в рамках одного процесса. Это практически исключает возможность исполнения таких программ – во всяком случае, в чистом виде - на многомашинных комплексах. Тем не менее, существуют вычислительные комплексы с большим количеством процессоров, предоставляющие разделяемую память, пригодную для исполнения многопоточных программ – так называемые NUMA-системы (Non-Uniform Memory Access, неоднородный доступ к памяти). Такие системы состоят из процессорных модулей, которые обычно имеют несколько процессоров и локальную память. Модули соединены между собой высокоскоростной коммутируемой или маршрутизуемой магистралью, которая обеспечивает доступ к ОЗУ других модулей. Разумеется, доступ к памяти другого модуля происходит медленнее, чем к локальному ОЗУ, но все-таки быстрее, чем копирование содержимого ОЗУ по сети. Многие реализации NUMA -CC-NUMA (Cache Coherent NUMA, NUMA с когерентным кэшем), COMA (Cache Only Memory Architecture) в том или ином виде реализуют локальное кэширование при межмодульном доступе к памяти.

NUMA системы с умеренным количеством процессоров (16, 32 или 64) применяются не только для вычислений, но и в качестве серверов баз

данных и серверов приложений и производятся серийно. Именно такую архитектуру имеют старшие модели серверов Sun Fire. Для вычислительных задач иногда применяются NUMA-системы с сотнями и тысячами процессоров. Межмодульная магистраль в таких системах обычно представляет собой маршрутизуемую сеть сложной (чаще всего гиперкубической) топологии. Такие машины делаются на заказ или небольшими сериями и, как правило, допускают установку дополнительных процессорных модулей. Именно такую архитектуру имел компьютер IBM DeepBlue, который в 1997 году выиграл матч у чемпиона мира по шахматам Г. Каспарова.

Наиболее распространенная технология разработки параллельных вычислительных программ с обменом сообщениями так и называется – MPI (Message Passing Interface, интерфейс передачи сообщений). MPI представляет собой библиотеку, реализующую высокоуровневый обмен типизованными сообщениями между процессами. Передача сообщений может осуществляться как по сети, так и через локальные средства IPC. Также предоставляется имитация семафоров, барьеров и разделяемой памяти при помощи сообщений.MPI обеспечивает запуск многопроцессных вычислительных приложений как на одном компьютере (даже однопроцессорном – это может быть полезно при отладке), так и на многомашинных вычислительных кластерах. Для большинства задач необходимо, чтобы эти машины были соединены высокопроизводительными сетевыми интерфейсами.

В последние годы возник интерес к гибридным приложениям MPI/OpenMP. Действительно, в современных вычислительных кластерах часто используются многопроцессорные машины. Кроме того, приложения MPI часто проводят довольно много времени в ожидании обмена данными; в это время машина могла бы заниматься исполнением другой части работы. На некоторых задачах использование гибридной вычислительной модели позволяет достичь улучшения на десятки процентов и даже в разы по сравнению с чистыми MPI и OpenMP.

Для задач, которыенуждаются в редком обменесообщениями небольшого объема, интересны вычислительные сети типа GRID. Такие сети представляют собой обычные персональные компьютеры, соединенные обычной офисной локальной сетью или даже через

Интернет. Эти сети часто допускают произвольное подключение и отключение машин. В этом случае, GRID система позволяет использовать персональные компьютеры во время простоя. Например, в научно-исследовательском учреждении во время рабочего дня компьютеры используются сотрудниками, а ночью или даже во время обеденного перерыва они подключаются к GRID и занимаются вычислениями. Разумеется, это возможно только если алгоритм допускает разбиение на совершенно автономные подзадачи, не нуждающиеся во взаимодействии друг с другом. Примерами таких задач являются уже упоминавшиеся взлом шифров методом грубой силы и вычисление интегралов методом Монте-Карло, а также поиск внеземных цивилизаций (Seti@Home), поиск генных последовательностей, многие задачи имитационного моделирования.

Правила приема задач и выставления оценок

Терминология

- Обычная задача – задачи с номерами от 1 до 28 включительно
- Задача "proxy" – задачи с номерами 29–31
- Эффективная задача – одна обычная задача или 1/3 задачи "proxy" (т.е. одна задача "proxy" эквивалентна 3 обычным задачам)

Собственно правила

Оценка за практикум выставляется на основе оценки за 4 семестр и количества задач, сданных в 5 семестре. Правила определения оценки таковы:

Оценка за 4 семестр	"отлично"	"хорошо"	"удовлетворительно"
"отлично"	17+2 "proxy"	11	6
"хорошо"	все задачи	11	6
"удовлетворительно"		16	6

По итогам практикума, студенты, получившие оценку 5, получают итоговую оценку 5 без сдачи экзамена ("автоматом"). Студенты, получившие за практикум оценку 4, обязаны сдавать экзамен и могут получить по результатам экзамена итоговую оценку от 2 до 5 включительно. Студенты, получившие за практикум оценку 3, обязаны сдавать экзамен и могут получить по результатам экзамена итоговую оценку от 2 до 4 включительно.

Студенты, не получившие оценки 3 за практикум, могут быть допущены к экзамену после собеседования и могут получить оценку не выше 3.

Требования к заданиям

Не допускается использование холостых циклов для синхронизации (если иное явно не оговорено заданием).

Трудновоспроизводимые "глюки" интерпретируются как ошибки соревнования. Обнаружив такое поведение, преподаватель обязан потребовать его устранения. Поскольку для трудновоспроизводимых ошибок безошибочный прогон и даже серия безошибочных прогонов программы не гарантирует отсутствия ошибки, студент должен не только устранить ошибку, но объяснить преподавателю ее причину. В том числе, при внесении исправлений в код программы, студент обязан объяснить, каким образом неисправленный код мог приводить к наблюдавшемуся поведению.

В случае неубедительного объяснения, преподаватель может требовать детализации объяснения вплоть до полного доказательства корректности применяемой схемы синхронизации. Преподаватель также имеет право требовать установки в коде программы дополнительных ожиданий (в том числе на случайные интервалы времени) и, напротив, требовать удаления существующих вызовов ожидания или межпоточной передачи управления.

Требования к заданиям "кэширующий proxy"

Прокси должен удовлетворять следующим требованиям:

1. Должен поддерживаться HTTP 1.0. Прокси должен корректно информировать как клиента, так и сервер об используемой версии протокола
2. Клиенты и серверы HTTP 0.9 (даже если таковые удастся найти) могут не поддерживаться, т.е. прокси может отказываться работать с ними, либо работа с ними может осуществляться в режиме HTTP 1.0. Возникающие при этом проблемы не являются основанием для отказа в приеме задания.
3. Поддержка HTTP 1.1 не обязательна. Поддерживать персистентные соединения 1.1 не требуется.
4. Необходимо поддерживать операцию GET. Поддержка всех остальных операций опциональна; отказ поддерживать остальные операции не является основанием для отказа в приеме задания. Ответы на операции PUT и POST, если сами эти операции будут реализованы, кэшировать не следует.
5. Кэшировать надлежит только ответы типа 200 (нормальная

передача страницы). Все остальные ответы следует передавать браузеру без изменений и кэшировать не следует.

6. Кэшировать следует как текстовые, так и бинарные ресурсы, с обязательным сохранением MIME типа ресурса.
7. Кэш хранится в памяти и может полностью теряться при перезапуске прокси.
8. Обработка полей заголовка, управляющих кэшированием, таких, как last modified и pragma no cache, не обязательна. Некорректная работа сайтов с динамическим HTML, обусловленная некорректной обработкой этих параметров, не является основанием для отказа в приеме задания.
9. Поддержка cookies не обязательна. Некорректная работа сайтов, использующих cookie (в том числе и для авторизации) не является основанием для отказа в приеме задания
10. Поддержка любых механизмов авторизации на сайтах не обязательна. Проверяется только корректность анонимного доступа.
11. Допускается как самостоятельная реализация анализа заголовка HTTP, так и использование third-party библиотек парсеров заголовка.
12. Допускается использование языков C и C++
13. Все варианты задания предполагают параллельную обработку запросов, т.е. при тестировании необходимо продемонстрировать возможность открыть несколько клиентских сессий и показать, что ни одна из сессий не ждет завершения операции ни одной из других сессий.
14. Псевдомногопоточная реализация и реализация с пулом потоков не должны иметь ограничений по количеству клиентских сессий (кроме количества дескрипторов открытых файлов). Т.е. обе эти реализации обязаны поддерживать больше сессий, чем имеется потоков в пуле. В этом смысле, псевдомногопоточную реализацию можно рассматривать как вырожденный случай пула потоков.
15. Допускается сдача пула потоков с объемом пула, равным 1, в качестве псевдомногопоточной реализации. В этом случае, преподаватель имеет право потребовать удалить из кода все примитивы pthread (возможно, при помощи конструкции #ifdef) и продемонстрировать сборку и работу приложения без использования libpthread.

16. Простая многопоточная реализация имеет право отвергать или задерживать входящие соединения при невозможности создать новый поток.
17. Если две клиентские сессии скачивают одну и ту же страницу, необходимо, чтобы обе сессии работали с одной и той же записью кэша, понимали, что запись неполная и адекватно реагировали на докачку.
18. Прокси должен корректно обрабатывать сброс клиентских сессий. В том числе, в случае, когда две или более сессий работали с одной записью кэша, после сброса одной из них, остальные сессии должны корректно продолжить докачку страницы.
19. Допускается создание двух нитей на каждое клиентское соединение: "клиентская" нить обрабатывает соединение с клиентом, "серверная" – с сервером. При работе двух клиентских сессий с одной записью кэша при этом следует создавать две клиентские нити, но одну серверную. При передаче клиенту полной записи кэша, серверную нить можно не запускать. В случае пула потоков, допускается разделение пула на два, серверных и клиентских нитей, исполняющих разный код.
20. При сбросе единственной сессии, работавшей с записью кэша, допускается как сброс докачки страницы и уничтожение записи в кэше, так и фоновое продолжение докачки. Преподаватель имеет право потребовать изменения стратегии обработки этой ситуации, т.е. потребовать переделать сброс докачки на ее фоновое продолжение или наоборот (это может быть полезно для проверки корректности управления записями в кэше).
21. Использование явных и неявных холостых циклов, а в особенности холостых циклов с ожиданием, для синхронизации потоков не допускается. Допускается использование стандартных примитивов синхронизации pthread, системных вызовов select и poll и вызовов асинхронного ввода-вывода.
22. В частности, высокая загрузка процессора (по показаниям top) при малой активности соединений интерпретируется как явный или неявный холостой цикл и является основанием для отказа в приеме задания. При обнаружении такого поведения, преподаватель имеет право потребовать доказательства корректности используемой схемы синхронизации.
23. Задержки при открытии сайтов интерпретируются как холостой цикл с ожиданием, и являются основанием для отказа в приеме

задания.

24. Вызовы функций sleep, usleep, pause и т.д. в коде приложения интерпретируются как холостой цикл с ожиданием и являются основанием для отказа в приеме задания.
25. Трудновоспроизводимые "глюки" при работе прокси интерпретируются как ошибки соревнования, и являются основанием для отказа в приеме задания. После обнаружения таких "глюков" преподаватель имеет право потребовать доказательства корректности используемой схемы синхронизации.
26. Допускается сдача подсистем прокси в качестве отдельных "простых" заданий. Студент сам должен понять, какие подсистемы могут быть сданы в качестве каких заданий. Необходимо продемонстрировать, что подсистема компилируется (возможно, с #ifdef'ами) и работает в виде самостоятельной программы. Сдача подсистем и целого прокси может производиться в любом порядке.

Задания практикума "Многопоточное программирование"

1. Создание нити

   Напишите программу, которая создает нить. Используйте атрибуты по умолчанию. Родительская и вновь созданная нити должны распечатать десять строк текста.

2. Ожидание нити

   Модифицируйте программу упр. 1 так, чтобы вывод родительской нити производился после завершения дочерней. Используйте pthread_join.

3. Параметры нити

   Напишите программу, которая создает четыре нити, исполняющие одну и ту же функцию. Эта функция должна распечатать последовательность текстовых строк, переданных как параметр. Каждая из созданных нитей должна распечатать различные последовательности строк.

4. Принудительное завершение нити

   Дочерняя нить должна распечатывать текст на экран. Через две секунды после создания дочерней нити, родительская нить должна прервать ее вызовом функции `pthread_cancel`.

5. Обработка завершения нити

   Модифицируйте программу упр. 4 так, чтобы дочерняя нить перед завершением распечатывала сообщение об этом. Используйте `pthread_cleanup_push`.

6. Многопоточный cp -R

   Реализуйте многопоточную программу рекурсивного копирования дерева подкаталогов, функциональный аналог команды `cp(1)` с ключом `-R`. Программа должна принимать два параметра –

полное путевое имя корневого каталога исходного дерева и полное путевое имя целевого дерева. Программа должна обходить исходное дерево каталогов при помощи `opendir(3C)/readdir_r(3C)` и определять тип каждого найденного файла при помощи `stat(2)`. Для определения размера буфера для `readdir_r` используйте `pathconf(2)` `(sizeof (struct dirent) + pathconf(directory)+1)`.

Для каждого подкаталога должен создаваться одноименный каталог в целевом дереве и запускаться отдельная нить, обходящая этот подкаталог. Для каждого регулярного файла должна запускаться нить, копирующая этот файл в одноименный файл целевого дерева при помощи `open(2)/read(2)/write(2)`. Файлы других типов (символические связи, именованные трубы и др.) следует игнорировать.

При копировании больших деревьев каталогов возможны проблемы с исчерпанием лимита открытых файлов. Очень важно закрывать дескрипторы обработанных файлов и каталогов при помощи `close(2)/closedir(3C)`. Тем не менее, для очень больших деревьев этого может оказаться недостаточно. Допускается обход этой проблемы при помощи холостого цикла с ожиданием (если `open(2)` или `readdir(3C)` завершается с ошибкой `EMFILE`, то допускается сделать `sleep(3C)` и повторить попытку открытия через некоторое время).

Обратите также внимание, что значения дескрипторов открытых файлов могут переиспользоваться, т.е. в разные моменты времени один и тот же дескриптор может указывать на разные файлы. Чтобы избежать связанных с этим проблем, избегайте передачи дескрипторов между нитями. Вся работа с дескриптором от создания до закрытия должна происходить в одной нити.

Дополнительное упражнение: при помощи команды `time(1)` сравните ресурсы, потребляемые вашей программой и командой `cp -R` при копировании одного и того же дерева каталогов. Объясните наблюдаемые различия. Каким образом их можно устранить? Следует ли вообще реализовать копирование файлов

таким способом и если да, то в каких условиях?

7. Вычисление Пи

   Напишите программу, которая вычисляет число Пи при помощи ряда Лейбница . Однопоточная версия такой программы доступна в файле `pi_serial.c`. Количество потоков программы должно определяться параметром командной строки. Количество итераций может определяться во время компиляции. Для передачи частичных сумм ряда, подсчитанных потоками, используйте `pthread_exit(3C)/pthread_join(3C)`.

   Обратите внимание, что на 32разрядных платформах `sizeof(double)>sizeof(void *)`, поэтому частичную сумму ряда нельзя преобразовывать к указателю, для нее надо выделять собственную память.

8. Вычисление Пи пока не надоест

   Модифицируйте программу упражнения 7 так, чтобы сама по себе она не завершалась. Вместо этого, после нажания ^-C (после получения сигнала `SIGINT`) программа должна как можно скорее завершаться, собирать частичные суммы ряда и выводить полученное приближение числа.

   Рекомендации: ожидаемое решение состоит в том, что вы установите обработчик `SIGINT`. Этот обработчик должен устанавливать глобальную флаговую переменную. Вычислительные нити должны просматривать значение флага через некоторое количество итераций, например через `1000000`, и выходить при помощи `pthread_exit`, если флаг установлен. Подумайте, как минимизировать ошибку, обусловленную тем, что разные потоки к моменту завершения успели пройти разное количество итераций. Скорее всего, такая минимизация должна обеспечиваться за счет увеличения времени между получением сигнала и выходом.

9. Обедающие философы

   Возьмите за основу программу `din_phil.c`. Эта программа

симулирует известную задачу про обедающих философов. Пять философов сидят за круглым столом и едят спагетти. Спагетти едят при помощи двух вилок. Каждые двое философов, сидящих рядом, пользуются общей вилкой. Философ некоторое время размышляет, потом пытается взять вилки и принимается за еду. Съев некоторое количество спагетти, философ освобождает вилки и снова начинает размышлять. Еще через некоторое время он снова принимается за еду, и т.д., пока спагетти не кончатся. Если одну из вилок взять не получается, философ ждет, пока она освободится. В программе `din_phil.c` философы симулируются при помощи нитей, периоды размышлений и еды – при помощи `usleep(3C)`, а вилки – при помощи мутексов. Философы всегда берут сначала левую вилку, а потом правую. При некоторых обстоятельствах это может приводить к мертвой блокировке. Измените протокол взаимодействия философов с вилками таким образом, чтобы мертвых блокировок не происходило.

10. Синхронизированный вывод

    Модифицируйте программу упр. 1 так, чтобы вывод родительской и дочерней нитей был синхронизован: сначала родительская нить выводила первую строку, затем дочерняя, затем родительская вторую строку и т.д. Используйте мутексы. Рекомендуется использовать мутексы типа `PTHREAD_MUTEX_ERRORCHECK`.

    Явные и неявные передачи управления между нитями (`sleep(3C)/usleep(3C)`, `sched_yield(3RT)`) и холостые циклы разрешается использовать только на этапе инициализации.

11. Синхронизированный вывод 2

    Докажите, что задача 10 не может быть решена с использованием двух мутексов без использования других средств синхронизации.

12. Синхронизированный вывод 3

    Решите задачу 10 с использованием условной переменной и минимально необходимого количества мутексов.

13. Синхронизированный вывод 4

    Решите задачу 10 с использованием двух семафоров-счетчиков

14. Синхронизированный вывод 5

    Если вы решили задачи 11 и 13, объясните, почему ваше доказательство неприменимо к семафорам-счетчикам.

15. Синхронизированный вывод двух процессов

    Решите задачу 10 с использованием двух процессов (а не нитей) и именованных семафоров-счетчиков

16. Синхронизированный доступ к списку

    Родительская нить программы должна считывать вводимые пользователем строки и помещать их в начало связанного списка. Строки длиннее 80 символов можно разрезать на несколько строк. При вводе пустой строки программа должна выдавать текущее состояние списка. Дочерняя нить пробуждается каждые пять секунд и сортирует список в лексикографическом порядке (используйте пузырьковую сортировку). Все операции над списком должны синхронизоваться при помощи мутекса.

17. Синхронизированный доступ к списку 2

    Переделайте программу упр. 16 так, чтобы с каждой записью (а также с заголовком списка) был связан свой собственный мутекс.

    Примечание: при перестановке записей списка, необходимой при реализации пузырьковой сортировки, необходимо блокировать мутексы трех записей.
    Примечание 2: чтобы избежать мертвых блокировок, мутексы записей, более близких к началу списка, всегда захватывайте раньше.
    Примечание 3: преподаватель может потребовать, чтобы программа включала две или более сортирующие нити, а также потребовать изменить интервал между сортировками.

18. Синхронизированный доступ к списку 3

Модифицируйте программу упр. 17 так, чтобы дочерняя нить засыпала на одну секунду между исполнениями каждого шага сортировки (между перестановками записей в списке). При этом можно будет наблюдать процесс сортировки по шагам.

19. Использование блокировки чтения-записи

    Модифицируйте программу упр. 16 так, чтобы вместо мутекса использовалась блокировка чтения-записи.

20. Использование блокировки чтения-записи 2

    Модифицируйте аналогичным образом программу упр. 18

21. Обедающие философы 2

    Решите задачу упр. 9 при помощи атомарного захвата вилок. Когда философ может взять одну вилку, но не может взять другую, он должен положить вилку на стол и ждать, пока освободятся обе вилки.

    Рекомендация: создайте еще мутекс `forks` и условную переменную. При попытке взять вилку философ должен захватывать `forks` и проверять доступность обоих вилок при помощи `pthread_trylock(3C)`. Если одна из вилок недоступна, философ должен освободить вторую вилку (если он успел ее захватить) и заснуть на условной переменной. Освобождая вилки, философ должен оповещать остальных философов об этом при помощи условной переменной. Тщательно продумайте процедуру захвата и освобождения мутексов, чтобы избежать ошибок потерянного пробуждения.

22. Производственная линия

    Разработайте имитатор производственной линии, изготавливающей винтики (`widget`). Винтик собирается из детали `C` и модуля, который, в свою очередь, состоит из деталей `A` и `B`. Для изготовления детали `A` требуется 1 секунда, B – две секунды, C – три секунды. Задержку изготовления деталей имитируйте при помощи `sleep`. Используйте семафоры-

счетчики.

23. Производитель-потребитель

Реализуйтеj очередь сообщений, которая может использоваться для обмена данными между двумя или большим количеством нитей. Реализация очереди должна поддерживать функции

```
void mymsginit(queue *);
void mymsqdrop(queue *);
void mymsgdestroy(queue *);
int mymsgput(queue *, char * msg);
int mymsgget(queue *, char * buf, size_t bufsize);
```

Допускается реализация на C++ с заменой `mymsginit` и `mymsgdestroy` на конструктор и деструктор, а операций `get` и `put` на соответствующие методы.

- `mymsgput` принимает в качестве параметра ASCIIZ строку символов, обрезает ее до 80 символов (если это необходимо) и помещает ее в очередь. Если очередь содержит более 10 записей, `mymsgput` блокируется. Функция возвращает количество переданных символов.
- `mymsgget` возвращает первую запись из очереди, обрезая ее до размера пользовательского буфера (если это необходимо). В любом случае, запись извлекается из очереди полностью. Если очередь пуста, `mymsgget` блокируется. Функция возвращает количество прочитанных символов.
- `mymsgdrop` должна приводить к разблокированию ожидающих операций `get` и `put`. Ожидавшие вызовы и все последующие вызовы `get` и `put` должны возвращать `0`.
- `mymsqdestroy` должна вызываться после того, как будет известно, что ни одна нить больше не попытается выполнять операции над очередью.

Необходимо продемонстрировать работу очереди с двумя производителями и двумя потребителями.

Для синхронизации доступа к очереди используйте семафоры-счетчики.

24. Производитель-потребитель 2

    Реализуйте задачу 23 с использованием условных переменных и минимально необходимого числа мутексов.

25. Многопоточный сервер

    Реализуйте сервер, который принимает TCP соединения и транслирует их. Сервер должен получать из командной строки следующие параметры:

    1. Номер порта `P`, на котором следует слушать.
    2. Имя или IP-адрес узла `N`, на который следует транслировать соединения.
    3. Номер порта `P'`, на который следует транслировать соединения.

    Сервер принимает все входящие запросы на установление соединения на порт `P`. Для каждого такого соединения он открывает соединение с портом `P'` на сервере `N`. Затем он транслирует все данные, получаемые от клиента, серверу `N`, а все данные, получаемые от сервера `N` – клиенту. Если сервер `N` или клиент разрывают соединение, наш сервер также должен разорвать соединение. Если сервер N отказывает в установлении соединения, следует разорвать клиентское соединение.

    Сервер должен обеспечивать трансляцию 510 соединений при лимите количества открытых файлов на процесс 1024. Сервер не должен быть многопоточным и никогда не должен блокироваться при операциях чтения и записи. Не следует использовать неблокирующиеся сокеты. Следует использовать `select` или `poll`.

26. псевдомногопоточный HTTP-клиент

    Реализуйте простой HTTP-клиент. Он принимает один параметр командной строки – URL. Клиент делает запрос по указанному

URL и выдает тело ответа на терминал как текст (т.е. если в ответе HTML, то распечатывает его исходный текст без форматирования). Вывод производится по мере того, как данные поступают из HTTP-соединения. Когда будет выведено более экрана (более 25 строк) данных, клиент должен продолжить прием данных, но должен остановить вывод и выдать приглашение `Press space to scroll down`.

При нажатии пользователем клиент должен вывести следующий экран данных. Для одновременного считывания данных с терминала и из сетевого соединения используйте системный вызов `select`.

27. псевдомногопоточный HTTP-клиент 2

    Реализуте задачу упр. 26, используя системные вызовы `aio_read`/`aio_write`.

28. Многопоточный HTTP-клиент

    Реализуйте задачу упр. 26, используя две нити, одну для считывания данных из сетевого соединения, другую для взаимодействия с пользователем.

29. Псевдомногопоточный кэширующий прокси

    Реализуйте простой кэширующий HTTP-proxy с кэшем в оперативной памяти.

    Прокси должен быть реализован как один процесс и один поток, использующий для одновременной работы с несколькими сетевыми соединениями системный вызов `select` или `poll`. Прокси должен обеспечивать одновременную работу нескольких клиентов (один клиент не должен ждать завершения запроса или этапа обработки запроса другого клиента).

30. Многопоточный кэширующий прокси

    Реализовать задачу 29, создавая для каждого входящего HTTP-соединения свою нить. При невозможности создать поток

допускается блокировать входящие соединения или возвращать ошибку.

31. Многопоточный кэширующий прокси с рабочими потоками

    Реализовать задачу 29, используя рабочие потоки (worked threads). При запуске прокси должен принимать параметр, целое число, указывающее размер пула потоков. Прокси должен запустить указанное число нитей. Необходимо обеспечить одновременную обработку количества запросов, превосходящего количество нитей в пуле; блокировка входящих соединений недопустима. Разумеется, при этом каждая из нитей в разные моменты времени будет вынуждена обрабатывать разные соединения. Для управления соединениями используйте `select` или `poll`.

Содержание

Титульная страница 2
Выходные данные 3
Лекция 1. Введение 4
Лекция 2. Реализации POSIX Threads API 15
Лекция 3. Создание и завершение нитей 23
Лекция 4. Атрибуты нитей и управление нитями 39
Лекция 5. Нити и стандартные библиотеки Unix 61
Лекция 6. Мутексы 74
Лекция 7. Блокировки чтения-записи, условные переменные, барьеры и семафоры-счетчики 91
Лекция 8. Мультиплексирование ввода/вывода и асинхронный ввод/вывод 102
Лекция 9. Событийно-ориентированные архитектуры 125
Лекция 10. Архитектуры многопоточных приложений 141
Лекция 11. Правила приема задач и выставления оценок 156
Лекция 12. Задания практикума "Многопоточное программирование" 161