Ловягин Никита Юрьевич

Основы проектирования операционных систем:

лабораторный практикум на xv6 под RISC-V

Учебное пособие

Санкт-Петербург
2024

Основы проектирования операционных систем:
лабораторный практикум на xv6 под RISC-V

(предварительная версия от 20.05.2024)

Предисловие

Учебная операционная система (ОС) xv6 создана в Массачусетском технологическом институте (MIT) и позиционируется как современная реализация версии 6 ОС UNIX. Операционная система разрабатывается для процессоров архитектуры RISC-V и может быть запущена на виртуальной машине QEMU на настольных (десктопных) и портативных (ноутбуках, нетбуках) компьютерах.

Пособие ориентировано на практическую часть общего курса по операционными системам. Главная цель данного пособия — на практике познакомить с принципам проектирования, реализации и функционирования основных компонентов ядра ОС: системными вызовами, управлением процессами, управлением памятью, обработкой прерываний, механизмами синхронизации, межпроцессным взаимодействием, файлами и файловыми системами, а также некоторыми особенностями программирования в пространстве ядра ОС.

Курс призван заложить базу для дальнейшего изучения проектирования и разработки компонентов и модулей ядра различных операционных систем для различных архитектур CPU, включая десктопные и серверные ОС на базе Linux, ОС для встроенных систем с архитектурой RISC-V, и другие. Курс может быть полезен системным и прикладным программистам для понимания особенностей работы и ограничений возможностей операционных систем. Кроме того, многие архитектурные и алгоритмические решения, используемые при проектировании и разработке операционных систем, могут применяться и в других задачах.

Первая глава посвящена краткому обзор теоретических основ ОС, основным понятиям и концепциям, таким как пространство ядра и пользовательское пространство, изоляция и системные вызовы; управление процессами, переключение контекста и обработка прерываний; механизмы синхронизации; файлы и файловые системы. Описываются особенности архитектуры процессоров, необходимые для поддержки задач ОС — режим ядра и пользовательский режим, аппаратные прерывания, трансляция адреса.

Во *второй главе* описываются базовые представления об архитектуре RISC-V и ее краткий обзор: регистры, адресация, машинные инструкции, режимы работы и уровни привилегий, страничная адресация, системные вызовы.

Третья глава посвящена устройству ОС xv6: описывается структура исходного кода и основные компоненты ядра; интерфейсы системных вызовов, прикладного программирования и программирования в пространстве ядра; средства управления процессами, синхронизацией и виртуальной памятью; файловая система и ввод/вывод.

В *четвертой главе* описываются получение, компиляция, запуск и отладка ОС xv6, создание программ на Ассемблере RISC-V и языке Си в пользовательском пространстве, добавление системных вызовов и компонентов ядра ОС, а также приводится ряд лабораторных заданий на программирование компонентов ядра и пользовательских программ на языке Си и Ассемблере RISC-V на различные темы (создание системных вызовов, обмен данными между пространством ядра и пользовательским пространством, переключение и синхронизация процессов, управление памятью, файлы и файловые системы) и указания путей их решения.

В начале каждой главы приводится список дополнительной литературы для дальнейшего изучения.

Оглавление

Глава 1. Введение

В главе дается краткий обзор теоретических основ проектирования и разработки ОС: основные понятия, алгоритмы, задачи, стоящие перед операционными системами. Поясняются такие понятия, как пространство ядра и пользовательское пространство, системные вызовы, изоляция, управление процессами, переключение контекста, обработка аппаратных прерываний, механизмы синхронизации, файлы и файловые системы. Описываются особенности архитектуры процессоров, необходимые для поддержки задач ОС: режим ядра и пользовательский режим, аппаратные прерывания, трансляция адреса. Так как данное пособие ориентировано на лабораторный практикум, оно не может заменить фундаментальный теоретический курс. Изучение теории операционных систем рекомендуется с использованием классических учебников.

1. Эндрю Таненбаум, Херберт Бос «Современные операционные системы», 4-е издание.
2. Эндрю Таненбаум, Херберт Бос «Современные операционные системы», 3-е издание (содержит описание систем реального времени, исключенное в 4 издании).

Предполагается также, что читатель книги знаком с основами программирования на языке Си, базовому использованию ОС семейства UNIX и командной строки, а также использованию системы контроля версий Git и средств сборки на базе GNU Compiler Collection (gcc), Debuger (gdb) и Make. Их изучение возможно в том числе по следующим источникам.

1. Брайан Керниган и Деннис Ритчи «Язык программирования Си».
2. Кристофер Негус «Библия Linux».
3. Мендел Купер «Искусство программирования на языке сценариев командной оболочки».
4. Ричард Столмен, Роланд Пеш, Стан Шебс и другие «Отладка с помощью GDB. Отладчик GNU уровня исходного кода».
5. Scott Chacon, Ben Straub «Pro Git» (распространяется свободно, в т. ч. русский перевод, доступно по адресу `https://git-scm.com/book/ru/v2`, дата доступа 01.02.2024).
6. Владимир Игнатов «Эффективное использование GNU Make».

§1.1. Задачи и абстракции операционных систем

Операционная система

Дать универсальное академическое определение операционной системы сложно, в том числе потому, что конкретные производители, относящие выпускаемыми ими на рынок программные продукты к операционным системам, могут включать в них различные компоненты, которые с теоретической точки зрения не являются системным программным обеспечением, или смешивать программы разных авторов в одном продукте. Например, долгое время в состав операционной системы Windows входили игры «Сапер» и «Косынка», но с академической точки зрения их следует рассматривать как прикладные программы; Linux является ядром операционной системы, но не собственно операционной системой, так как не содержит прикладных программ для работы пользователя: в дистрибутивах ОС на базе Linux используются соответствующие программы из проекта операционной системы GNU и другие.

С другой стороны, давая определение операционной системы в начале учебного курса пришлось бы неизбежно сослаться на ранее неизвестные понятия. Например, это определение может выглядеть таким образом.

Операционная система *— это программное обеспечение, управляющее компьютером, его аппаратными и программными ресурсами, распределяющее их между программами и пользователями, предоставляющее более удобный, функциональный и безопасный интерфейс для пользователей и прикладных программистов, чем предоставляемый непосредственно аппаратным обеспечением.*

Для того, чтобы понять, что имеется в виду под этим определением, скорее всего, нужно предварительно изучить курс по операционным системам. Поэтому сначала рассмотрим наглядный пример того, какую работу берет на себя операционная система компьютера, и на его основе сформулируем основные *задачи*, стоящие перед операционными системами. Этот набор задач и можно рассматривать как определение операционных систем (ОС).

В качестве первой задачи обратим внимание на такое обстоятельство. Чаще всего ***системное программное обеспечение***, к которому обычно относят не только сами ОС с дополнительными программами, но и ***встроенное программное обеспечение*** главных плат, контроллеров различных устройств и компьютеров, обычно противопоставляется ***прикладному программному***

обеспечению, ориентированному непосредственно на решение тех задач, которые стоят перед пользователем компьютера, т. е. задач той предметной области, где пользователь применяет компьютер — бытовых, производственных и др. Поэтому сразу можно сказать, что одна из основных задач операционной системы — *предоставление пользователю интерфейса для запуска прикладных программ*. В следующих пунктах данного параграфа рассмотрим другие задачи ОС на примерах.

Файл как пример абстракции ОС

Рассмотрим в общих чертах как устроена работа с данными, хранимыми в файле. Пусть прикладная программа, например, видеоплеер, открывает файл. Предположим, что прикладной программист написал для этого действия такую строку на языке Си (или аналогичную на другом языке высокого уровня):

```
FILE *in_file = fopen (path_to_file, "r");
```

Пользователь выбрал фильм для просмотра, после чего данная строчка, точнее, соответствующий ей двоичный исполняемый код, будет выполнен компьютером.

В строке содержится вызов функции `fopen` — функции стандартной библиотеки языка Си. Эта функция умеет работать с файлами, что является удобным для прикладного программиста. В то же время аппаратные ***накопители информации*** (устройства хранения данных), на которых этот файл реально сохранен, ничего о файлах не знают, а могут только прочитывать и записывать блоки данных фиксированного размера, например 512 байтов или 4 килобайта, с физического носителя информации. То есть файл является ***программной абстракцией***, а не аппаратным объектом.

Файл, размер которого обычно даже не кратен размеру физического блока, должен быть размещен (распределен) по блокам на накопителе. За это отвечает ***файловая система (file system) данного накопителя***. Существуют различные файловые системы, которые решают эту задачу по-разному. Они имеют свои преимущества и недостатки — разный набор дополнительных возможностей, разную скорость работы, разные ограничения на размер и длину имени файла, разное качество и надежность работы, могут быть универсальны или специфичны к виду накопителя и решаемым задачам.

Существуют различные накопители информации, взаимодействующие с компьютером по различным ***аппаратным интерфейсам***: например, накопители на жестких магнитных дисках и твердотельные накопители с интерфейсом SATA, сменные накопители с интерфейсом USB, твердотельные

накопители с интерфейсом PCI-E и др. Все эти интерфейсы и накопители устроены по-разному и требуют передачи различных команд для выполнения операций ввода/вывода.

Таким образом, во-первых, работа с файлами программируется *проще*, чем взаимодействие по аппаратным интерфейсам. Файл, как поименованная упорядоченная совокупность данных, является абстракцией более высокого уровня, более близкой к прикладным задачам и пользователю, чем блок данных.

Во-вторых, файл является более *универсальным*, чем накопитель информации. Более того, файл может располагаться не на физическом носителе, подключенном к данному компьютеру, а быть доступным, например посредством компьютерной сети — лежать в общей сетевой папке. Прикладным программам (и их пользователям) неважно, где располагается этот файл, какой аппаратный интерфейс у устройства хранения данных или интерфейс доступа к общему сетевому ресурсу, по которому компьютер прочитывает и сохраняет данные этого файла, какая именно файловая система используется на этом накопителе.

Кроме того, регулярно появляются новые файловые системы, новые типы накопителей данных и новые типы интерфейсов, но при этом ранее написанные программы получают возможность работать с ними без изменения как исходного, так и двоичного кода — для этого достаточно обеспечить поддержку устройства в ОС. Уже хотя бы для обеспечения этой независимости и универсальности, решение задачи взаимодействия с накопителем информации и размещением на нем данных нельзя вынести ни на уровень прикладных программ, ни на уровень библиотеки языка программирования высокого уровня, иначе для каждого нового случая пришлось бы писать отдельный код если не в самой программе, то в библиотеке каждого языка программирования с постоянным обновлением.

В-третьих, файл является более *безопасным* инструментом, чем накопитель данных. С одним компьютером могут работать несколько пользователей, которые хотят обеспечить сохранность своих личных или служебных секретов — для этого ***пользователям*** и ***администраторам системы*** разграничивают ***права доступа*** к файлам. При каждом действии с файлом должна проводится проверка, действительно ли пользователь имеет право на эту операцию (в текущем примере — право просмотра данного видео, т. е. открытия файла и чтения его содержимого). Эту проверку бессмысленно выполнять на уровне прикладных программ и библиотек языков программирования, так как иначе продвинутый пользователь мог бы без особого труда обойти все огра-

ничения, просто написав свою программу — ведь накопители информации и аппаратные интерфейсы ничего о пользователях и их правах доступа не знают.

Еще одна проблема состоит в том, что когда у пользователя запущено несколько программ, их несогласованные действия могут привести — не по злому умыслу и даже не из-за ошибки программирования, а именно из-за нескоординированности действий в состоянии *гонки* — к порче или потере данных.

Например, программа A хочет дописать `10` байтов в конец файла `1`, размер которого составляет `1346` байтов. Для этого она прочитывает блок накопителя, где содержатся последние $1346 - 2 \cdot 512 = 322$ байта данных (оставшиеся `190` байтов не используются, т. е. содержат «мусорные» данные — это, увы, необходимо, так как для обеспечения удаления и дозаписи файлов файловые системы обычно отводят под каждый файл целое количество блоков), дополняет его соответствующими данными. Далее она должна сохранить его обратно на накопитель данных. Одновременно (на другом процессоре или вытеснив исполнение программы A и получив процессорное время) программа B делает почти то же самое: прочитывает блок, в котором все еще содержатся `322` байта концовки файла `1`, дополняет своими `20` (!) байтами и сохраняет измененный блок. Далее снова работает программа A — и сохраняет измененный своими силами блок. Таким образом, 20 байтов, записанные программой B, уже пропали безвозвратно.

Далее программа A прочитывает блок, где содержится фактический размер файла (его необходимо хранить на накопителе, так как реальный размер не равен количеству блоков, умноженному на 512), ставит новый по ее мнению размер файла — `1346+10=1356` — и сохраняет его. Но следом программа B тоже изменяет размер файла по своему мнению — `1346+20=1366`. После этого данные программы B не просто оказались утерянными, а сам файл пришел в неконсистентное состояние: последние `10` байтов этого файла содержат «мусорные» данные (которые были на накопителе, прочитаны программой A и записаны обратно).

В данном случае файл является ***общим ресурсом*** для программ A и B, что является источником проблемы. Для решения такой проблемы необходимо запретить одновременный доступ к общим ресурсам (в данном случае к блокам данных, отвечающим за структуру файла) — сначала одна программа должна получить монопольный доступ к ресурсу и выполнить свою работу, а затем другая. Соответствующее согласование совместной работы невозможно выполнить исключительно на уровне прикладных программ и библиотек без

риска порчи данных, поэтому решение проблем, связанных с использованием общих ресурсов разными приложениями ложится на операционную систему.

Кроме того, само пространство хранилища данных может рассматриваться как общий аппаратный ресурс, распределяемый между пользователями: администратором может быть введено ограничение на максимальный объем данных, хранимых пользователем на компьютере (***квотирование ресурсов***) во избежание злоупотреблений. Таким образом, перед операционной системой стоит задача *управления ресурсами компьютера*,

Подытоживая данный пример, можно сказать, что *файл, как программная абстракция, создается именно на уровне операционной системы, предоставляющей необходимый интерфейс прикладного программирования — более удобный, универсальный и безопасный, чем аппаратный — и управляющий программными и аппаратными ресурсами компьютера. Именно в этом и состоят основные задачи операционной системы.*

Основная задача ***файлов*** как абстракции и интерфейса работы с ними — обеспечить хранение и доступ к *упорядоченным последовательностям байтов, идентифицируемым посредством уникального имени*, а файловых систем — организовать размещение данных и метаданных (атрибутов, прав доступа и т. д.) файлов собственно на накопителях информации. Говоря более точно, доступ к файлам на уровне ОС обеспечивается несколькими абстракциями различных уровней.

- Собственно физические устройства хранения (накопители данных) могут разбиваться на части (***разделы***). Разделы могут состоять из физически смежных блоков или быть чисто логическими разделами, размещенными произвольно на одном или нескольких физических устройствах хранения (в этом контексте используется также термин «том»). Могут также использоваться виртуальные накопители данных (например, виртуальные приводы компакт-дисков) или виртуальные разделы — отсутствующие физически, но представленные в виде файл-образа, в котором сохраняются данные виртуального устройства. Данные виртуальных устройств также могут сохраняться в памяти. Все эти физические и логические (виртуальные) *устройства обеспечивают чтение и запись адресуемых блоков данных*, поэтому в Linux называются ***блочными устройствами*** (***block device***, в противоположность ***символьным устройствам*** — ***character device***, обеспечивающим ввод/вывод последовательностей байтов). Таким образом, *блочные устройства — программная абстракция над физическими накопителями данных*. В контексте других ОС такой термин

может не использоваться. Так как исторически первыми получившими широкое применение накопителями данных, обеспечивающими блочный ввод/вывод, являлись накопители на магнитных дисках, в некоторых случаях может использоваться исторический термин «***диск***» (логический и физический) — как синоним соответствующей программной абстракции, безотносительно того, является ли физический носитель данных диском или нет. Для обеспечения работы физического, логического или виртуального блочного устройства в ОС должна иметься поддержка такого типа устройства. Часть операционной системы, обеспечивающая поддержку работы устройства, обычно называется ***драйвером***.

- Файловые системы блочных устройств отвечают за расположение ***данных*** и ***метаданных*** файлов (прав доступа, меток времени доступа к файлу и его изменения, вспомогательных атрибутов, служебных данных файловой системы и других) по блокам. Примерами таких файловых систем служат ext4 и btrfs, используемые в ОС на базе Linux. Помимо файловых систем блочных устройств, существуют файловые системы, не привязанные к ним, но также организующие хранение и доступ к файлам каким-либо способом (например, в оперативной памяти — tmpfs, по сети — NFS и SSHFS и другим). *Файловые системы, отвечающие за доступ и размещение данных на конкретных типах коллекций, по адресам доступа или на блочных устройствах, и являются программными абстракциями над способом доступа к самим данным* (блочным вводом/выводом, доступом по сети и т. д.) называются ***конкретными файловыми системами***. Для обеспечения доступа к файлом конкретных файловых систем в ОС должен иметься драйвер соответствующей файловой системы. По указанным выше причинам, ФС блочных устройств также могут называться дисковыми файловыми системами. Не связанные с блочными устройствами — «не-дисковые» ФС — называют сетевыми, распределенными, временными, виртуальными и т. д., в зависимости от способа доступа к данным
- Для предоставления приложениям унифицированного доступа к файлам конкретных файловых систем, ОС должна реализовать еще один слой абстракции над ними. Конкретные экземпляры файловых систем конкретных коллекций данных (блочных устройств или иных конкретных адресов и способов доступа к коллекциям данных) должны подключаться в единое адресное пространство файлов: адресами файлов и каталогов являются их полные пути. В Linux этот слой аб-

стракции называется ***виртуальной файловой системой*** (***VFS***, ***Virtual File System***), а акт подключения конкретного экземпляра файловой системе к конкретному каталогу — ***монтированием***.

Операционная система предоставляет ряд других программных абстракций. В следующих пунктах рассмотрим основные их них.

Процессы и потоки выполнения

Процесс (***process***) — *абстракция ОС, содержащая один или несколько параллельных* ***потоков выполнения (thread)****. Поток выполнения, в свою очередь, представляет собой набор последовательно исполняемых команд (машинных инструкций).*

В первом приближении под процессом можно понимать запущенную программу. Это не является точным определением, так как, во-первых, пользователь может запустить несколько процессов одной программы, а во-вторых, одна программа может создать несколько процессов. Однако здесь подчеркивается то, что программа является статическим, постоянно существующим объектом (набор файлов, данных), а процесс — динамическим, запущенным и исполняемым в некоторый момент времени.

Чаще всего процессы ***изолированы*** друг от друга и от операционной системы для обеспечения безопасности: они не имеют прямого доступа к физической оперативной памяти, а работают в ***виртуальном адресном пространстве***, т. е. используют виртуальные адреса, ***транслируемые в физические адреса*** по правилам, устанавливаемым ОС. Это позволяет исключить чтение и изменение данных и кода других процессов, в т. ч. процессов других пользователей и процессов самой ОС, что позволило бы обойти все ограничения на права доступа, например, к файлам.

Потоки одного процесса (если их несколько) в свою очередь исполняются в одном виртуальном адресном пространстве, что делает их, например, удобным инструментом ***параллельного программирования*** — они могут решать одну задачу одновременно на разных процессорах. Современные ОС, как правило, реализуют поддержку многопоточных процессов в том числе для того, чтобы обеспечить возможность параллельной работы в рамках одной программы.

В простейшем случае операционная система поддерживает только однопоточные процессы, т. е. в каждом процессе есть ровно один поток выполнения. Это относится, в частности, к старым версиям операционных систем, выпускавшимся для однопроцессорных (одноядерных) компьютеров. По этой исторической причине, а также в целях упрощения изложения, в классиче-

ской литературе и курсах по ОС часто рассматриваются именно однопоточные процессы и их работа на однопроцессорных компьютерах. Поэтому иногда эти понятия на каком-то уровне изложения отождествляются, или под процессом понимается сам процесс и его единственный поток. Это относится, в частности, и к ОС xv6. Современные ОС чаще всего проектируются с поддержкой многопоточных процессов, поэтому в первой и второй главах данного пособия, посвященных описанию общих принципов проектирования ОС, эти понятия разделяются.

В большинстве операционных систем процессы рассматриваются именно как ***единицы изоляции*** *(каждый работает в своем изолированном виртуальном адресном пространстве в памяти) и заявки на некоторых ресурсы разделяемые между потоками этого процесса (например, открытые файлы)*, в то время как потоки рассматриваются как ***единицы работы*** — *заявки на выполнение некоторого кода (последовательности машинных инструкций) процессором*.

Как правило, исполняемый код (последовательность машинных инструкций, получаемая в результате трансляции исходного кода программы) у потоков одного процесса общая, но они работают с разными данными. Таким образом, потоки одного процесса могут исполнять одни и те же участки кода над разными данными, выполнять разные ветви одних и тех же условных конструкций или даже запускать разные функции — но фактически выполнять одну и ту же программу. Точнее говоря, каждому процессу в памяти выделяется несколько непересекающихся блоков — ***сегментов***: общий для всех потоков процесса ***сегмент кода***, содержащий исполняемый код, общий для всех потоков и всех функций процесса ***сегмент данных***, содержащий глобальные и статические данные процесса, а также персональный для каждого потока ***сегмент стека***, содержащий локальные данные для каждого экземпляра вызова функции в каждом потоке (***стековый фрейм***).

Один процессор может исполнять только одну последовательность инструкций. Он не осведомлен ни о процессах, ни о потоках. При наличии в компьютере нескольких процессоров (или многоядерного процессора) возможно одновременное исполнение нескольких потоков одного или различных процессов — ***одновременная многопоточность***. В этом случае для создания параллельных многопоточных приложений требуется поддержка многопоточности со стороны операционной системы — распределение потоков по процессорам.

На однопроцессорных компьютерах поддержка многопоточности со стороны ОС не требуется. Однако даже в этом случае на уровне ОС может быть

реализована ***многозадачность*** — поддержка одновременной работы нескольких приложений, т. е. одновременного выполнения нескольких однопоточных процессов. Отметим, что под одновременностью работы здесь понимается именно субъективное восприятие пользователем ситуации, когда несколько программ выполняют свою работу в одно и то же время на достаточно длительном промежутке времени, хотя на самом деле в каждый конкретный момент процессор исполняет только один поток. Более того, часто в современных ОС число запущенных (одновременно работающих) потоков превышает число имеющихся процессоров (ядер). То есть многозадачность ОС на однопроцессорных компьютерах может рассматриваться как частный случай одновременного выполнения потоков, когда их (потоков) количество превышает количество процессоров.

Для обеспечения одновременной работы потоков в количестве, превышающем количество доступных процессоров, ОС должна производить переключение процессора с исполнения одного потока на исполнение другого потока. Таким образом, от многозадачной ОС требуется распределение такого ресурса как ***процессорное время*** между потоками выполнения, а от ОС с поддержкой многопроцессорных компьютеров (одновременной многопоточности) — еще и распределение потоков по процессорам. Достаточно частое переключение и создает субъективное восприятие пользователем одновременной работы нескольких задач.

Распределение процессорного времени между потоками обеспечивает ***логический*** (виртуальный, абстрактный) ***параллелизм*** (***виртуальную многопоточность*** — в противоположность одновременной многопоточности) в рамках одного процессора. Поэтому проблемы параллельного программирования — выстраивание очередности работы с общими ресурсами — всегда стоят перед ОС, код которой обрабатывает запросы от одновременно работающих потоков.

Виртуальная память

Физическая оперативная память (ОЗУ, RAM) является дефицитным ресурсом и ее может не хватать для хранения данных и кода всех запущенных процессов. В то же время хранение именно *всех данных всех запущенных программ* в ОЗУ может быть избыточным. Потребление памяти и реальное использование данных приложениями сильно зависит от действий программиста и пользователя. Например, программа может выделить из кучи некоторый массив данных

```
int *a = malloc (big_data * sizeof (*a));
```

и даже инициализировать его, но достаточно долго не использовать. Это может являться нормальной работой программы, и не рассматриваться как ошибка разработки. Например, этот массив нужен для отображения вкладки в браузере, которую пользователь оставил неактивной, или содержит изображение, которое уже выведено на экран и долго не требует перерисовки, которая, однако, может понадобиться в любой момент. В этом случае процесс совершенно корректно не делает вызов `free`, поэтому выделенная память закреплена за ним, но не используется и не может быть передана другим приложениям. И это не говоря о ситуации, когда ошибки программирования приводят к утечке памяти: выделенный блок остается закрепленным за процессом, но не будет использоваться никогда.

Выходом из ситуации является сохранение долго неиспользуемых данных процесса из оперативной памяти, например, на долговременный накопитель данных, распределение освободившейся памяти между другими процессами, и восстановление данных в физической оперативной памяти в тот момент, когда процесс к ним обратится. Это организует ***виртуальную оперативную память*** *— абстракцию ОС над оперативной памятью, в которой* ***прозрачным образом*** *(т. е. без дополнительных действий со стороны программиста и информирования приложений) размещаются данные процессов, размер которой может превышать доступный объем физической памяти.*

Примеры других абстракций ОС

Абстракциями ОС также являются, например ***сетевые сокеты***, предоставляющие приложениям доступ к сети, независящий от сетевого устройства и типа подключения (проводного, беспроводного, мобильного), ***аудиомикшер***, обеспечивающий вывод звука на различные устройства и его совмещение от одновременно работающих приложений, ***очередь печати***, позволяющая приложениям не общаться с различными типами принтеров, локальными и сетевыми, напрямую, но и исключающая наложение печатаемых из различных источников документов на одном листе бумаги, ***аккаунты пользователей***, для которых обеспечивается ***аутентификация*** (подтверждение личности, подлинности, например, с помощью логина и пароля), и ***авторизация*** (предоставление прав на определенные действия), и другие.

Основные функции ОС

Обобщая вышесказанное, к базовым функциям ОС следует отнести такие задачи.

- ***Абстрагирование (виртуализация) оборудования***: предоставление программного интерфейса для доступа к аппаратному обеспечению компьютера — программных абстракций и виртуальных ресурсов (например, файлы, виртуальная память, процессы, потоки и их логический параллелизм и т. д.). Этот интерфейс должен быть более удобным (простым в программировании, чем аппаратный), унифицирующим разнообразные аппаратные интерфейсы и безопасным (обеспечивающим разграничение прав доступа и защиту от некорректного использования). ***Обеспечение безопасности*** работы пользователей и программ может быть выделено в самостоятельную задачу, стоящую перед ОС.
- ***Управление программными и аппаратными ресурсами компьютера***: их распределение между программами и пользователями, организация совместной работы пользователей и программ без взаимных помех и риска приведения системы и ресурсов в некорректное состояние.
- ***Предоставление базового пользовательского интерфейса*** для запуска прикладных программ и работы с данными пользователей.

§1.2. Процессоры: аппаратная поддержка функций ОС

Заметим, что задачи обеспечения безопасности не могут быть решены *только* на уровне ОС. Действительно, если бы прикладные программы имели прямой доступ к аппаратному обеспечению — накопителям информации, физической оперативной памяти и др., то все ограничения на права доступа можно было бы обойти, просто изменив нужный блок данных или двоичный код ОС в памяти. Для того, чтобы закрыть доступ к определенным аппаратным функциям компьютера, необходима соответствующая ***аппаратная поддержка***.

Архитектура набора машинных команд и работа процессора

Процессор *— как* ***центральное процессорное устройство (CPU)*** *компьютера, так и процессоры контроллеров и компонентов компьютерных систем, например,* ***графические процессорные устройства*** *(****GPU****) — это аппаратное устройство, выполняющее* ***машинные инструкции (команды)***.

Машинные инструкции кодируются в двоичном виде. Двоичный исполняемый файл программного обеспечения (как прикладного, так и системного) содержит в себе в т. ч. последовательность инструкций, которые должны быть исполнены. Различные типы процессоров поддерживают различный набор машинных инструкций, называемых ***архитектурой набора машинных команд процессора*** (***ISA, instruction set architecture***).

Наиболее известные и активно используемые в настоящее время архитектуры набора машинных команд следующие:

- **x86-64** (также известная как **AMD64** для процессоров, выпускаемых фирмой AMD или **EM64T** для процессоров, выпускаемых фирмой Intel) — преобладающая архитектура в персональных стационарных и портативных компьютерах и серверах. Ранее эту нишу занимала архитектура **IA-32**.
- **z/Architecture** — архитектура мейнфреймов, серверов и суперкомпьютеров фирмы IBM.
- **POWER** (**PowerPC**, **Power ISA**) — архитектура некоторых персональных компьютеров, совместная разработка Apple, IBM и Motorola (альянс AIM).
- **ARM** — преобладающее семейство архитектур на мобильных устройствах (смартфонах, планшетах) и встраиваемых системах с низким энергопотреблением. В настоящее время используется версия **ARMv8**, вытесняющая версию **ARMv7**. Разработчиком архитектуры является ARM Holdings, процессоры выпускают такие фирмы как Apple, Qualcomm, Samsung, MediaTek и другие.
- **SPARC** — используется в серверах компании Oracle.
- **MIPS** (различные модификации) — используется во встраиваемых системах, сетевом оборудовании, например, роутерах (маршрутизаторах) и др.
- **RISC-V** — относительно новая открытая и свободная архитектура, разрабатываемая RISC-V International по инициативе исследователей Калифорнийского университета (Беркли). Расширяемость и открытость спецификаций делает возможным и привлекательным создание процессоров на данной архитектуре различными фирмами в различных целях, что делает ее перспективной для широкого распространения и использования.

Все процессоры выполняют вычисления и управляются с помощью ***регистров*** — сверхбыстрых ячеек памяти (особых, аппаратно расположенных

непосредственно в ядре процессора), и адресуемых специальными кодами машинных инструкций. Таким образом, архитектура машинных команд включает в себя не только собственно перечень команд (инструкций), но и перечень регистров процессора данной архитектуры. Этот перечень различен для различных архитектур, но обычно выделяются следующие группы регистров.

- ***Арифметические регистры***, предназначенные для хранения чисел (целых) и операций над ними, то есть собственно производства арифметических и логических вычислений.
- ***Адресные регистры***, предназначенные для хранения адресов в оперативной памяти (указателей). К ним могут относиться собственно регистры, предназначенные для хранения адресов (значений указателей), определенных во время работы программы, а также, например, ***регистр указателя вершины стека*** — позиции в памяти, относительно которого процессор отсчитывает адреса локальных данных вызываемых функций программ. Значение данного регистра изменяется на соответствующую величину при помещении данных в стек вызовов и удалении данных из него; процессор автоматически вычисляет адрес значения в стеке по заданному смещению относительно вершины стека. Эти смещения рассматриваются как адреса локальный данных функции — они жестко прописываются в двоичном коде, но благодаря изменению значения регистра, содержащего адрес вершины стека, для каждого вызова функции ячейка памяти, соответствующая этим данным, будет своя.
- ***Регистры общего назначения***, которые могут использоваться и как арифметические, и как адресные.
- ***Константные регистры***, хранящие некоторое неизменное значение, например 0. Значение такого регистра не может быть изменено.
- Регистры для хранения данных различных типов: ***чисел с плавающей точкой***, ***векторные регистры***, хранящие массивы чисел, и другие.

  Обычно арифметические, адресные и другие регистры данных могут быть свободно прочитаны и изменены программой с помощью соответствующих инструкций.
- ***Указатель инструкции***, содержащий адрес следующей машинной инструкции, подлежащей исполнению процессором. Этот регистр изменяется процессором автоматически на длину кода машинной инструкции после ее исполнения и может быть изменен программно с

помощью специальных инструкций, например, инструкций условного и безусловного перехода.

- ***Сегментные регистры***, содержащие адреса блоков памяти, содержащие код, данные, стек вызовов или иной участок кода или данных программы. Относительно значений данных регистров отсчитывается адрес, содержащийся в указателе инструкций, стека и др. (например, адрес значения в стеке может вычисляться исходя из значения регистра сегмента стека, указателя вершины стека и смещения относительно вершины стека, а адрес инструкции, например, для условного или безусловного перехода — исходя из значения регистра сегмента кода и смещения относительно начала сегмента кода). Таким образом, в двоичном коде программ жестко прописываются относительные адреса (данных, кода), но одновременно запущенные процессы могут получать разные значения сегментных регистров и обращаться к разным участкам памяти, даже если эти относительные адреса совпадают. Эти регистры обеспечивают на уровне процессора поддержку существования у каждой запущенной программы нескольких сегментов различного назначения, но не обязаны обеспечивать изоляцию виртуального адресного пространства процесса, хотя и могут являться частью этого механизма.
- ***Регистры управления, статуса и другие***, отвечающие за работу CPU, режимы и уровни привилегий, управление работой с памятью, трансляцию адресов и изоляцию процессов, обработку ошибок и исключительных ситуаций, права доступа приложений и другие архитектурно-зависимые функции.

Таким образом, работу процессора можно описать следующим образом.

- Процессор вычисляет адрес данных или инструкции в памяти исходя из заданного значения — постоянного (жестко прописанного в двоичном исполняемом коде) или переменного (указателя, содержащегося в адресном регистре или в указателе инструкций). Указанное значение может использоваться напрямую или рассматриваться как смещение относительно некоторого регистра (указателя стека, начала сегмента и др.). Конкретный способ преобразования зависит от архитектуры и используемой инструкции. В дальнейшем полученный адрес может быть подвергнут трансляции, описанной на стр. 26.
- Инструкция, адрес которой определяется исходя из значения указателя инструкций, извлекается из памяти и исполняется. Инструкция типично может осуществлять

- перемещение данных из памяти в регистр или обратно;
- запись значения в память по адресу или в регистр;
- арифметические или логические вычисления;
- помещение данных в стек или их удаление (приводящие также к изменению значения указателя стека);
- условный или безусловный переход по заданному адресу в памяти;
- вызов функции по указанному адресу, возврат из функции и некоторые другие действия, например, *вызов функций операционной системы* — ***системный вызов*** — речь о котором пойдет на стр. 26;
- взаимодействие с устройством ввода/вывода (в частности, с накопителями информации, сетью, клавиатурой, мышью, графической картой и т. п.);
- и другие действия.

Если инструкция предусматривает передачу управления (переход) по новому адресу — условный или безусловный переход, вызов функции и др. — то процессор изменяет значение указателя инструкции на соответствующее значение. В противном случае значение указателя инструкций увеличивается на размер исполненной инструкции в байтах.

- Инструкции исполняются автоматически, одна за другой. Однако, в этот процесс может вмешаться механизм аппаратных прерываний, речь о которых пойдет на стр. 30.

Размер регистров обычно составляет несколько байтов (типично от 1 до 8, исключение — векторные регистры). При этом размер адресного регистра (а значит и размер адреса, который может быть задан в машинной инструкции) определяет максимальный теоретический объем оперативной памяти, который может адресовать программа. Так, для 32-разрядного регистра этот размер составляет 4 гигабайта (2^{32} байта), а для 64-разрядного — 16 эксабайтов (2^{64} байта), хотя такой объем памяти многократно превышает реализуемый на данный момент в реальном оборудовании, а существующие процессоры также имеют аппаратный лимит на размер адреса, существенно меньший данного значения.

Под ***разрядностью архитектуры процессора*** понимается разрядность регистров (арифметических, адресных). Разрядность адреса (адресной шины, шины памяти) может совпадать с разрядностью адресных регистров или от-

личаться от нее (например, полный адрес может задаваться двумя регистрами). Архитектуры IA-32 и ARMv7 являются 32-разрядными, x86-64 и ARMv8 — 64-разрядными, архитектура RISC-V имеет как 32-разрядную, так и 64-разрядную модификацию.

Приведем пример ***псевдокода на языке псевдоассемблера некоторой псевдоархитектуры***, показывающий изменение значения регистра-указателя инструкции при исполнении программы в виде таблицы. Данный код написан в иллюстративных целях и не соответствует никакой реальной архитектуре процессора. В первом столбце указан (виртуальный) адрес начала инструкции — некоторый номер байта (шестнадцатеричный), во втором — шестнадцатеричный дамп машинной инструкции (как он мог бы выглядеть), в третьем — текстовое представление этой инструкции, в четвертом — ее пояснение, в пятом — значение указателя инструкций *после* исполнения данной инструкции с пояснением. Предполагается, что изначальное значение указателя инструкции 0x100, поэтому будет исполняться *вторая* строка этой таблицы — до нее тоже может присутствовать некоторый код, здесь не показанный. Пусть регистры R1 и R2 — арифметические.

Адрес	**Машинный код**	**Мнемокод**	**Пояснение**	**Значение указателя инструкций**
…	…	…	…	…
`0x0100`	`1A 01 02`	`ADD R1 R2`	Добавить к значению регистра R1 значение регистра R2	0x103 (размер инструкции — 3 байта)
`0x0103`	`0B 01 00 00 00 01`	`SUB R1 0x1`	Вычесть из значения регистра R1 число 1	0x103 (размер инструкции — 6 байтов)
`0x0109`	`00 00 00 02 00`	`JMP 0x200`	Передать управление по адресу 0x200	0x200 (безусловный переход)
`0x011D`	…	…	В данном примере эти инструкции пропускаются	
`0x0200`	`0C 01 02`	`SUB R1 R2`	Вычесть из значения регистра R1 значение регистра R2	0x203 (размер инструкции — 3 байта)
`0x0203`	…	…	…	…

Для пояснения прямой и косвенной адресации рассмотрим следующий пример. Пусть структура данных программы в памяти следующая

Адрес	Машинный код	Значение	Пояснение
…	…	…	…
0x0500	00 00 00 00	0	Целое число, значение 0, 4 байта
0x0504	48 65 6C 6C 6f 2C 20 77 6F 72 6C 64 0A 00 00 00	'Hello, world\n'	Нуль-терминированная (ASCIIZ) строка длины 13 байтов, в буфере размера 16 байтов
0x0514	20	' '	Символ «пробел» (ASCII-код 20)
0x0515	…	…	…

Пусть регистр A1 — адресный и его значение — 0x504 (шестнадцатеричное), т. е. он указывает на *третью* строку участка данных в памяти. Тогда инструкции чтения-записи данных по адресу могут иметь следующий вид и смысл.

Мнемокод	Пояснение
LOADWORD R1 [0x500]	Загрузить в регистр R1 значение длины 4 байта (машинное слово) из памяти по абсолютному адресу 0x500 — там сейчас число 0.
LOADADDR A2 0x504	Загрузить в регистр A2 значение адреса (пусть длина адреса 32 бита) 0x504 — адрес начала строки 'Hello, world\n'.
SAVEWORD R2 A1 (-4)	Сохранить значение регистра R2 длины 4 байта (машинное слово, если разрядность регистра больше, то обычно сохраняются младшие 4 байта) по адресу в памяти 0x500 (значение регистра A1 минус 4 байта) — перезаписать значение 0, имеющееся там.
LOADBYTE R3 A1 (0x10)	Загрузить в регистр R3 значение длины 1 байт (обычно в младшие байты регистра) из памяти по адресу 0x514 (значение регистра A1 плюс 0x10 байтов) — 0x20 (код символа «пробел»).

Таким образом, с помощью косвенной адресации можно организовать как адресацию относительно начала блока данных программы, зафиксировав значение адресного регистра, а также стек программы (стек вызовов) с помощью изменения адресного регистра на нужную величину — размер фрейма (***stack frame***) данных функции. Также относительная адресация может использоваться для указания адреса перехода.

Изоляция процессов и пространства ядра: режим ядра, системные вызовы, трансляция адреса

Как уже было отмечено, обеспечить безопасность на уровне защиты прав доступа только программными средствами невозможно, так как, во-первых, если процесс *может* исполнить любую процессорную инструкцию, то он *не обязан* обращаться к абстракциям операционной системы, а может, например, осуществить ввод и вывод напрямую, а во-вторых, при наличии доступа к физическим адресам оперативной памяти, процесс может прочитать или изменить данные других процессов и операционной системы. Таким образом, при отсутствии аппаратной защиты, сохраняется возможность прочитать или изменить конфиденциальные данные других пользователей на накопителе данных или в памяти, подменить права доступа к файлу, изменить код операционной системы, отвечающий за проверку прав доступа — то есть совершить целенаправленную атаку или просто нарушить работу других программ или операционной системы в целом в результате ошибки в коде программы.

Для решения этой проблемы в процессорах предусматривается как минимум два режима работы с разным ***уровнем привилегий*** (***privilege levels***) — ***режим ядра*** *(**kernel mode**), также называемый привилегированным режимом или режимом супервизора и* ***пользовательский режим (user mode)***. *В режиме ядра могут быть выполнены все машинные инструкции, в пользовательском режиме — только их безопасное подмножество, разрешенное к использованию прикладными программами. Таким образом, все машинные инструкции подразделяются на* ***пользовательские*** *и* ***привилегированные****, доступные только из режима ядра.*

Режим ядра предназначен для исполнения исключительно ***ядра операционной системы*** *(**operating system kernel**) — компонента ОС, имеющего полный контроль над аппаратурой компьютера и программными абстракциями ОС.* В некотором смысле определение режима ядра процессора и ядра операционной системы тавтологично: ядро ОС — код, исполняемый в режиме ядра процессора, режим ядра процессора — режим, предназначенный для исполнения кода ядра ОС. Но с включением в определение режима ядра процессора возможности исполнения подмножества машинных инструкций, объявленных привилегированными, тавтология разрешается.

В ***пользовательском режиме*** *недоступно исполнение машинных инструкций, обеспечивающих прямое взаимодействие с аппаратурой: ввод и вывод, обращение к физической оперативной памяти, изменение защищен-*

ных регистров, регулирующих режим работы и трансляцию адреса, и других параметров конфигурации работы процессора. Таким образом, инструкции и регистры процессора подразделяются на пользовательские (доступные и из пользовательского режима и из режима ядра) и привилегированные (доступные только из режима ядра).

Обратим внимание, что несмотря на схожесть терминов, понятие пользовательского режима и режима ядра (привилегированного режима, режима супервизора) не связано с пользователями и администраторами (супервизорами) ОС: программы, запущенные от имени администратора ОС, в любом случае исполняются в пользовательском режиме процессора. Также режим ядра (***kernel***) процессора не связан с понятием «ядра» (***core***) многоядерной (многопроцессорной) архитектуры процессора. Каждое ядро процессора (или даже каждый поток выполнения процессора, если архитектура позволяет параллельное выполнение нескольких потоков на одном ядре) может независимо находиться в режиме ядра или в режиме пользователя и переключаться между ними. Каждое ядро также имеет свой независимый набор регистров, т. е. с точки зрения программной архитектуры ядро является отдельным процессором. То же самое относится к процессорам, ядра которых могут исполнять несколько параллельных потоков — операционной системой и прикладными программами это воспринимается как существование соответствующего количества независимых процессоров.

Для выполнения операции, требующей привилегий режима ядра, например, для открытия файла, системная или прикладная программа, работающая в пользовательском режиме, должна обратиться к ядру ОС с запросом на выполнение такой операции. Делается это при помощи ***системного вызова (system call)*** *— вызова функций ядра ОС.* Строго говоря, именно поток выполнения пользовательского процесса должен сделать системный вызов: не сами пользователи напрямую, а запущенные ими программы — потоки их выполнения — открывают файлы, вводят и выводят данные и совершают другие манипуляции. Сам акт запуска приложения — создание процесса и потока — тоже производится посредством системного вызова из некоторого другого процесса, например, процесса оболочки ОС. Самый первый процесс и поток его выполнения создается при старте операционной системы.

Системные вызовы похожи на вызов функций внутри программы или функции библиотеки из программы, но требует для это *специальной машинной инструкции — процессор должен и переключить режим работы (повысить уровень привилегий), и передать управление соответствующему*

участку кода ядра ОС в памяти, чтобы только код ядра мог исполняться в режиме ядра.

Ядро ОС, получив управление посредством системного вызова, делает проверку наличия у процесса права на осуществление данной операции, например, наличие права на чтение открываемого для чтения файла. Точнее говоря, права доступа определяются исходя из того, какой процесс сделал данный системный вызов, но обычно настраиваются для пользователей, запускающих процессы, или иным способом, в соответствии с конфигурацией политики безопасности и механизмом ее обеспечения в данной ОС,

При наличии такого права ОС выполняет необходимые действия и возвращает результат работы системного вызова пользовательскому потоку (например, дескриптор открытого файла), а при отсутствии — соответствующую ошибку. По завершении работы системный вызов с помощью специальной инструкции возвращает управление потоку пользовательского приложения. Исполняя эту инструкцию, процессор и переключается в пользовательский режим, и передает управление в точку, где находится инструкция, следующая за системным вызовом.

Необходимо учитывать, что системный вызов — сравнительно ресурсоемкая операция, требующая (помимо непосредственно осуществления полезной работы) вызова функции, переключения режима процессора, и выполнения инструкций режима ядра для проверки прав и возможностей выполнения операции, обмена данными между приложением и ядром и т. д.

Как уже было отмечено, пользовательский процесс не должен иметь доступа к физической оперативной памяти — ему доступны только адреса в виртуальном адресном пространстве данного процесса (в виртуальной памяти). *Теоретически виртуальное адресное пространство при некоторых допущениях можно было бы организовать посредством системных вызовов, но так как обращение к оперативной памяти — одна из наиболее частых операций, выполняемых программами в ходе своей работы (любое исполнение инструкции предполагает загрузку ее кода из памяти; сами инструкции могут осуществлять загрузку данных из памяти и сохранения данных в память), такая реализация замедлила бы работу компьютера в несколько десятков раз.*

Для реализации виртуального адресного пространства процесса предусматривается аппаратный механизм ***трансляции адреса*** *— автоматического преобразования* ***виртуального адреса*** *процесса в физический адрес в соответствии с правилами, установленными операционной системой для данного процесса.* Правила записываются в оперативную память в виде специальных

таблиц дескрипторов сегментов или страниц. С помощью сегментных регистров и регистров управления, защищенных от изменения в пользовательском режиме, происходит настройка используемых данным процессом правил. В дескрипторах также может быть указано, в каком режиме привилегий должен исполняться код данной области памяти. Конечно, на трансляцию адреса также тратится время, но оно многократно меньше времени, необходимого на системный вызов. Кроме того в процессорах применяются различные аппаратные механизмы ускорения, кеширования и буферизации данной процедура.

При ошибке трансляции адреса, например, использовании потоком процесса некорректного виртуального адреса (указателя) или отсутствия правила для преобразования, процессор сгенерирует ***исключительную ситуацию, ошибку*** — передаст управление ядру операционной системы. ОС должна проверить причины возникновения данной ситуации. *Если это ошибка в пользовательском процессе, то он принудительно завершается, а если это следствие того, что данные из оперативной памяти выгружены в долговременную, операционная система должна подгрузить эти данные, внести правки в таблицы трансляции и дать процессору возможность продолжить исполнение потока.* Таким образом, виртуальная память, как абстракция операционной системы, как механизм отделения виртуального адресного пространства процессов и увеличения объема доступной памяти сверх физической, реализуется посредством корректной настройки правил преобразования адресов для ядра и каждого отдельного процесса.

При запуске операционной системы процессор работает в режиме ядра с отключенным механизмом трансляции адреса (***реальный режим***). Операционная система, работая с физическим адресами, настраивает таблицы трансляции адреса. Физические адреса этих таблиц заносятся в специальные регистры, изменение которых доступно только в режиме ядра. После этого включается режим трансляции адреса (***защищенный режим***). ОС выделяет каждому процессу свою таблицу трансляции — записывает свое значение данного регистра. После этого передается управление процессу с переключением в пользовательский режим. Соответственно, пользовательскому процессу недоступно изменение данного регистра и содержимого таблицы трансляции, т. е. он не может выйти за пределы своего виртуального адресного пространства.

С помощью механизмов трансляции адреса и введения уровней привилегий обеспечивается ***изоляция пространства ядра (kernel space)*** *и* ***пользовательского пространства (user space)*** *— областей виртуальной памяти, в*

которых размещаются, соответственно, код и данные ядра и пользовательских процессов. Каждый процесс, в свою очередь, исполняется в отдельном виртуальном адресном пространстве.

Прерывания и исключения, аппаратные прерывания, прерывания по таймеру

Решение, при котором процессор может только исполнять поток инструкций автоматически и непрерывно, имеет ряд недостатков.

Во-первых, при таком архитектурном решении операционной системе приходится периодически сканировать все устройства ввода/вывода на предмет совершенной операции, например, нажатия пользователем клавиши или движения мышью. Это возможно, но требует дополнительных ресурсов, что критично на маломощных системах при значительном количестве устройств. Современные USB-устройства (в т. ч. клавиатура и мышь) используют именно такой подход, однако во многих случаях более оптимальным является *автоматическая передача управления операционной системе по факту ввода/вывода* (данный подход используют, например, клавиатуры и мыши, работающих по более старому интерфейсу PS/2, а также многие современные устройства ввода/вывода, такие как накопители данных, сетевые устройства и др.).

Во-вторых, если поток выполнения в течение продолжительного времени не будет выполнять системный вызов вследствие осуществления длительных вычислений или программной ошибки (зацикливания, зависания), то операционная система долго или вообще никогда не получит управление. В этом случае будет невозможно (особенно на однопроцессорной системе) ни переключить задачу для реализации многозадачности, ни прервать работу «зависшей» программы по команде пользователя. Поэтому *требуется механизм, автоматически, периодически или по истечении тайм-аута, передающий управление операционной системе*.

В-третьих, во время работы процессора возможно возникновение различных ошибок и исключительных ситуаций, *требующих прерывания исполнения последовательности инструкций и реакции со стороны операционной системы*. Примерами таких ситуаций являются

- попытка исполнения двоичной последовательности, не являющейся корректной машинной инструкцией;
- попытка исполнения привилегированной инструкции в пользовательском режиме;

- ошибка обращения к памяти при трансляции адреса, выхода в недоступную область, нарушения защиты области памяти;
- целочисленное деление на ноль и другие.

Для решения этих проблем в процессорах предусмотрен ряд механизмов, которых можно обобщить под термином ***прерывание*** *или* ***ловушка*** *(**trap**)*. ***Прерывание*** *— это механизм приостановки исполнения текущей последовательности инструкций (потока выполнения) с передачей управления* ***обработчику прерываний*** *— специальной функции ядра ОС, выполняющей быструю реакцию на произошедшее событие и, в зависимости от результатов обработки, либо возвращающую управление прерванному потоку, либо завершающую его.* В зависимости от архитектуры процессора выделяют несколько видов прерываний, называемые разными терминами, которые могут различаться с идейной точки зрения или в технических тонкостях реализации, и не всегда однозначно используемые в литературе. Рассмотрим основные типов механизмов.

- ***Аппаратные прерывания*** (***interrupt***, ***hardware interrupt***) **—** *прерывания, инициируемые внешними устройствами (аппаратурой, устройствами ввода/вывода).* Обеспечивают быструю реакцию ОС на события ввода/вывода. Технически осуществляются с помощью специального устройства — ***контроллера прерываний***, обнаруживающего сигнал прерывания от устройства ввода/вывода и сообщающего об этом процессору. Каждому устройству сопоставляется номер прерывания (IRQ, ***Interrupt ReQuest*** *—* ***запрос прерывания***), по которому ОС может определить, какое именно устройство вызвало прерывание. На современных компьютерах контроллер прерываний обычно является компонентом процессора или главной платы (хотя и может быть отдельным внешним устройством), а также является программируемым (настраиваем) средствами ОС (например, позволяет настроить номер и приоритет прерываний для различных устройств).
- ***Прерывания по таймеру*** *— частный случай аппаратных прерываний, генерируемых специальным устройством,* ***таймером****, с определенной частотой (например, 1000 раз в секунду) или по заданному тайм-ауту.* Не связан с вводом/выводом, но *позволяют ОС в нужный момент исполнить свой код и осуществить управление процессами и потоками, в т. ч. переключение между задачами.*
- ***Исключения (expection), ошибки (fault)*** *— нефатальные и фатальные ошибки, возникающие в ходе выполнения инструкций, требующие реакции операционной системы (деление на ноль, отсутствие*

физического адреса для запрашиваемого виртуального, нарушения защиты и др.). Являются исходом исполнения процессором очередной инструкции и требуют реакции со стороны ОС — устранения нефатальной ошибки и обеспечения продолжения работы процесса, или его принудительного завершения, если ошибка фатальна.

- ***Программные прерывания (software interrupt)*** *— прерывания, генерируемые с помощью соответствующих машинных инструкций.* Отличаются от аппаратных тем, что инициируются не аппаратурой, а самой выполняющийся программой (потоком).
- ***Системные вызовы*** *— в некоторых архитектурах системные вызовы реализованы с помощью программных прерываний или с помощью отдельных инструкций.* Действительно, подобные ситуации также решаются с помощью передачи управления ядру ОС и возврата управления потоку в пользовательском пространстве, что роднит их с программными прерываниями и исключениями в смысле реализации.

Аппаратные прерывания отличаются от других ловушек тем, что они происходят ***асинхронно***, т. е. независимо от того, какие инструкции исполняет текущий поток. В то же время ошибки, исключения, программные прерывания и системные вызовы являются прямым следствием исполняемой инструкции.

Аппаратные прерывания обычно реализуются и обрабатываются следующим образом. Каждому устройству сопоставляется номер аппаратного прерывания IRQ. В специальной таблице, расположенной в оперативной памяти, каждому номеру сопоставлен адрес, по которому передается управление при возникновении данного прерывания — адрес обработчика прерывания. Эта таблица заполняется операционной системой, ее адрес фиксирован или указывается в специальном привилегированном регистре процессора (пользовательский процесс не может изменить содержимое этой таблицы и этого регистра). При возникновении аппаратного прерывания процессор автоматически сохраняет в стеке или в специальном привилегированном регистре адрес инструкции, на которой произошло прерывание, повышает уровень привилегий и передает управление по указанному адресу, после чего выполняется код обработчика прерываний в режиме ядра ОС. По завершении обработки прерывания, обработчик должен вызывать специальную машинную инструкцию — возврат из прерывания. При исполнении данной инструкции процессор автоматически понижает уровень привилегий и передает управление по сохраненному адресу инструкции.

Обработка других ловушек происходит аналогичным образом. При возникновении ошибки или исключения причина ошибки автоматически записывается процессором в специальный привилегированный регистр. При системном вызове процесс (точнее, один из его потоков) самостоятельно записывает номер системного вызова и его аргументы в требуемый интерфейсом данной ОС непривилегированный регистр. Процессор автоматически передает управление по зарегистрированному ОС адресу. Операционная система может завершить данный процесс аварийно (при фатальной ошибке) или по запросу (если процесс сделал системный вызов завершения своей работы), вернуть ему результаты обработки (выходные данные системного вызова), сохранив их в регистре, и продолжить исполнение следующей инструкции, либо дать возможность исполнить инструкцию, вызвавшую ошибку (например, если данные были подгружены ОС из виртуальной памяти в физическую). После этого ОС выполняет специальную инструкцию завершения обработки ловушки.

Строго говоря, ОС не обязана после завершения обработки ловушки вернуть управление тому же потоку, который исполнялся при возникновении этого события — управление может быть передано другому потоку того же или даже другого процесса. Действительно, процесс может быть завершен, поток может быть быть замещен потоком того же или другого процесса принудительно или перестать претендовать на процессорное время по каким-либо внутренним причинам — в ожидании завершения другого процесса, освобождения доступа к общему ресурсу, завершения операции ввода/вывода (которая может осуществляться устройством из памяти без траты процессорного времени). В этом случае ОС должна произвести переключение на другой поток того же или другого процесса, заполнив регистры таким образом, чтобы вызов инструкции возврата из обработчика привел к передаче управления новому потоку.

Процессы и потоки: контекст и переключение контекста

Процессор не осведомлен о процессах и потоках выполнения. Даже при наличии одного ядра и однопоточного процессора многозадачными операционными системами реализуется ***виртуальная многопоточность*** — *процессам предоставляются небольшие* (десятки миллисекунд) *кванты* ***процессорного времени*** *поочередно* (не обязательно равные и не обязательно строго в порядке очереди: порядок и выделяемая доля времени определяется заданными правилами, приоритетом процессов и алгоритмом распределения процессорного времени), *в результате чего создается ощущение одновре-*

менной работы приложений. В реальных современных операционных системах суммарное число выполняющихся потоков всех процессов часто превышает количество процессоров (ядер, параллельных потоков, которые может исполнять процессор), поэтому перед ОС все равно стоит задача распределения процессорного времени — переключения между потоками как одного, так и различных процессов. В последнем случае также происходит переключением между процессами.

Для процессора поток выполнения представлен значением регистров пользовательского пространства, в т. ч. арифметических регистров, указателя инструкции и указателя стека. Процесс, как единица изоляции, представлен регистрами, отвечающими за используемую оперативную память и трансляцию виртуальных адресов в физические, обычно привилегированными. Регистры, отвечающие за трансляцию адреса, задают виртуальное адресное пространство процесса и являются общими для потоков одного процесса.

Для переключения между потоками операционной системе требуется сохранить значения всех регистров текущего потока и заменить значениями регистров нового потока, а затем передать управление по соответствующему адресу нового потока с помощью специальной или стандартной инструкции переключения, перехода или возврата из прерывания. Данный процесс называется ***переключением контекста потока (context switch)****.*

Под ***контекстом (context) потока*** *понимается совокупность данных, необходимая для выполнения данного потока.* В процессоре — это значения регистров, в т. ч. указатель инструкции, в оперативной памяти — данные, код и стек потока, а в ОС — различная идентификационная и служебная информация, относящаяся к данному потоку (эта информация сохраняется ОС в оперативной памяти, в пространстве ядра, обычно в специальной таблице). При переключении контекста происходит приостановка исполнения текущего потока и продолжение работы другого приостановленного потока с точки приостановки.

При переключении контекста между потоками разных процессов происходит также переключение ***контекста процесса***. В контекст процесса, помимо контекста всех его потоков, включается его изолированное виртуальное адресное пространство, представленное соответствующими регистрами процессора, и все данные в нем, а также регистрационные данные процесса в ОС, установленные права доступа, открытые файлы и другие элементы.

Заметим, что для переключения контекста на системах с ***общей памятью***, т. е. системах, где все процессоры (ядра) имеют доступ к одному и тому же физическому адресному пространству, необходимо и достаточно сохра-

нить текущие значения регистров потока и/или процесса и загрузить сохраненные значения регистров другого (ранее приостановленного) потока и/или процесса. Сами данные потока и процесса (в т. ч. содержимое стека, код процесса, данные кучи, глобальные данные потока, глобальные данные процесса и т. д.), хотя и могут рассматриваться как часть контекста, не требуют сохранения и восстановления при переключении — достаточно изменить соответствующие адресные регистры процессора. Для систем с ***распределенной памятью***, где у каждого процессора имеется доступ к своей собственной памяти (например, компьютерные кластеры), для переключения контекста требуется передача всех данных процесса с одной памяти на другую. Это требует существенно больше времени, чем простое переключение контекста путем изменения значения регистров в системах с общей памятью.

Переключение контекста прозрачно для потоков и процессов — они «не замечают», что их исполнение было приостановлено (исполнялся другой поток), а затем продолжено, возможно даже на другом процессоре. Если поток A был прерван на какой-то инструкции, все значения его регистров были сохранены, потом какое-то время исполнялся поток B, а затем в процессор снова были загружены сохраненные регистры потока A (в т. ч. указатель инструкции), то работа потока A возобновится с точки прерывания его работы автоматически (разве что поток может обнаружить длительный «провал» во времени, но обычно такое не нужно).

Системный вызов не производит переключение контекста с точки зрения архитектуры ядра — код системного вызова в ядре исполняется в контексте вызвавшего его потока. Это необходимо в том числе для возможности проверки прав доступа у процесса данного потока и обменом данными между процессом (пользовательским пространством) и пространством ядра. Аналогичная ситуация с ошибками, исключениями и программными прерываниями — ОС должна иметь возможность определить что именно и в каком именно потоке пошло не так и как на это реагировать. Аппаратные прерывания, инициируемые внешними устройствами, де-факто могут исполняться в контексте прерванного потока, хотя там он не имеет особого смысла — поток прерывается асинхронно и не ждет взаимодействия с ОС и данных от нее. Однако для самого акта переключении в режим ядра и исполнения кода ядра, в т. ч. доступа к виртуальному адресному пространству ядра — данным ядра в памяти — требуется соответствующее изменение адресных регистров процессора, т. е. ***переключение на контекст ядра потока, или на контекст обработки прерывания (аппаратного)***. Кроме того, обычно во избежание проблем, вызванных ошибками в коде пользовательского приложения, и недопущения нарушения изоляции пространства ядра и

пользовательского пространства, в ядре используется ***стек потока пространства ядра***, отличный от стека потока пользовательского пространства. Поэтому при системном вызове адресный регистр стека также подлежит изменению. У каждого потока должен быть свой стек пространства ядра, в то время как стек обработчиков всех аппаратных прерываний может быть общим.

Кроме того, процессор (в зависимости от архитектуры) не обязан сохранять, изменять и восстанавливать значения пользовательских регистров при срабатывании ловушки. Обработчик прерываний, системных вызовов и других ловушек, в свою очередь, может изменить их значения в ходе своей нормальной работы — а чтобы поток смог корректно продолжать работу, эти значения необходимо восстановить при возврате из обработчика. Таким образом, ОС должна сохранять регистры пользовательского пространства потока и восстанавливать их при практически любой обработке ловушки, а при переключении контекста потока (процесса) — сохранять регистры одного потока (процесса) и загружать регистры другого потока (процесса).

Порядок байтов

Важно обратить внимание на тот факт, *что различные архитектуры процессоров используют различный* ***порядок байтов*** (***endianness***). Порядок байтов относится к расположению в памяти байтов чисел — как тех, которые являются данными программы, так и тех, которые являются составной частью кода машинных инструкций.

Рассмотрим 32-битное число 16909060, или 0x01020304 в шестнадцатеричной системе. Оно занимает в памяти 4 байта, содержащие числа 0x01, 0x02, 0x03, 0x04 соответственно. При прямом порядке байтов они будут расположены от старшего к младшему, т. е. как перечислены в обычной человекочитаемой записи числа, а при обратном — от младшего к старшему, то есть 0x04, 0x03, 0x02, 0x01.

Результат выполнения такого Си-кода:

```
uint32_t n = 0x01020304;
char *v = (char *)&n;

for (int i = 0; i < 4; ++i)
    printf ("%0x02X ", v[i]);
printf ("\n");
```

зависит от порядка байтов процессора. Для прямого порядка это

```
0x01 0x02 0x03 0x04
```

а для обратного будет выведено

```
0x04 0x03 0x02 0x01
```

Это не является ни неопределенным поведением, ни поведением, зависимым от компилятора. Результат строго определяется порядком байтов архитектуры процессора.

Большинство современных архитектур, включая x86-64 и RISC-V использует ***обратный*** (***little-endian***) ***порядок байтов***, в то время как ряд ранних архитектур (например SPARC), используют ***прямой*** (***big-endian***) ***порядок байтов***. Некоторые архитектуры (POWER, MIPS) предоставляют возможность переключения или реализации обоих вариантов порядка байтов.

Прямой порядок байтов удобен для чтения дампов памяти человеком, в то время как обратный порядок проще в реализации и имеет некоторые удобства при программировании. Например, если в память записать число большой разрядности, реальная запись которого занимает меньшее количество байтов (например, число 21 (0x15) в uint32_t будет представлено в памяти как `15 00 00 00`), то при прочтении его как числа меньшей разрядности будет прочитано корректное значение (значение uint16_t прочитает из памяти `15 00`, т. е. 0x15, uint8_t прочитает `15`, т. е. 0x15 и т. п.). При прямом порядке байтов в таких случаях будет получено нулевое значение.

Порядок байтов следует обязательно учитывать при сохранении и передаче двоичных данных. При разработке операционных систем это относится, в частности, к передаче данных по сети и сохранении метаданных файлов и файловых систем на накопителе данных. В противном случае это приведет к некорректной интерпретации числовых данных при прочтении или получении их на компьютере с другой архитектурой. Поэтому во всех этих случаях следует при записи (передаче) приводить данные к какому-то фиксированному порядку байтов, а при прочтении конвертировать в реальный порядок байтов текущей архитектуры. Также порядок байтов должен быть установлен в аппаратном интерфейсе (взаимодействием с устройствами ввода/вывода, шиной подключения устройств и т. д.).

§1.3. Проектирование ОС: архитектура и алгоритмы

Уровни абстракции и архитектура ОС

В операционных системах можно выделить следующие компоненты, которые можно рассматривать как уровни абстракции.

1. ***Обработчики прерываний*** — наиболее близкие к аппаратуре процедуры. Должны быть короткими (быстро отрабатывающими и завершающимися) процедурами, в т. ч. потому, что внутри обработчика аппаратных прерываний обычно запрещаются аппаратные прерывания по крайней мере от того же устройства (в противном случае это привело бы к прерыванию работы обработчика и повторному входу в этот же обработчик, что как минимум существенно усложнило бы его программную реализацию из-за рисков рекурсии и гонки, и привело бы к ухудшению производительности). Должны исполняться в пространстве ядра, так как непосредственно общаются с устройствами ввода/вывода.
2. ***Драйверы устройств*** — специфичное для конкретного аппаратного устройства программное обеспечение, осуществляющее взаимодействие ОС и данного устройства. Помимо драйверов конкретных аппаратных устройств существуют драйверы других типов, например, ***драйверы файловых систем,*** а также ***драйверы виртуальных устройств*** (например, виртуальных дисководов для компакт-дисков или виртуальных сетевых карт), не существующих физически, но имитирующих работу реальных устройств, ***драйверы логических устройств*** (например, разделов накопителя данных), созданных поверх физических, ***драйверы псевдоустройств*** (например, драйвер для устройств null и urandom в Linux), осуществляющие операции ввода/вывода особым образом, не привязанным к конкретному устройству или его имитации. Драйверы аппаратных устройств могут также включать в себя обработчики прерываний. Остальная часть кода драйвера в зависимости от архитектуры ОС может исполняться как в пространстве ядра, так и в пользовательском пространстве.
3. ***Слой абстракции аппаратного обеспечения*** — слой программных абстракций над конкретной аппаратурой, обеспечивающий унифицированный программный интерфейс для доступа к многообразному аппаратному интерфейсу оборудования. Взаимодействует с оборудованием посредством драйверов устройств.
4. ***Интерфейс системных вызовов*** — набор системных вызовов, предоставляемый ОС. Может зависеть от архитектуры процессора. На двоичном уровне системный вызов обычно идентифицируются по уникальному номеру, на уровне исходного кода используются символические имена.

5. ***Интерфейс прикладного программирования*** — набор библиотечных функций, представляющих собой обертки над системными вызовами, работающие в пользовательском пространстве. Обычно унифицированы для всех архитектур, на которые портирована данная операционная система. Именно эти функции должны вызываться прикладными программами для обеспечения их переносимости. Данный интерфейс является специфичным для каждой ОС, но обычно независим от архитектуры процессора (если ОС поддерживает несколько архитектур). Эти функции могут вызываться как системными, так и прикладными программами напрямую. В этом случае исходный и двоичный код этих программ будет зависимым от ОС. Функции библиотек языков программирования высокого уровня также вызывают данные функции — реализация библиотеки языка программирования является зависимой от ОС. Программы, использующие только стандартную библиотеку, являются независимыми от ОС по крайней мере на уровне исходного кода.
6. ***Управляющие компоненты ОС,*** в частности, ***планировщик задач (потоков, процессов)***, отвечающий за создание, завершение, выбор очередности исполнения потоков и процессов, ***диспетчер памяти***, отвечающий за распределение физической памяти между процессами и поддержку виртуальной памяти, и другие. Эти компоненты ОС непосредственно отвечают за управление ее работой и исполнением приложений, распределением ресурсов между процессами и т. д. Обычно являются частью ядра ОС. Обычно не предоставляют слой программных абстракций для прикладных программ, но управляют их работой и ресурсами, поэтому от качества их реализации сильно зависит качество и скорость работы компьютера в целом.

Типы архитектуры ядра ОС подразделяются на ***монолитную*** и ***микроядерную***. ***Монолитное ядро*** *представляет собой единую программу, работающую в пространстве ядра и берущую на себя все задачи операционной системы: управление, абстрагирование оборудования, драйверы устройств. При* ***микроядерной архитектуре*** *как минимум драйверы устройств работают как независимые программы в пользовательском пространстве.* Иногда выделяют ***гибридные*** *архитектуры, при которых некоторые компоненты ОС исполняются в пользовательском пространстве, но драйверы устройств исполняются в пространстве ядра.*

К преимуществам монолитной архитектуры относятся высокая скорость работы, т. к. взаимодействие между компонентами реализуется как простой

вызов функции, в то время как микроядру требуется больше переключений между пользовательским режим и режимом ядра — переключение контекста связано с накладными расходами. В то же время монолитное ядро сложнее изменять в процессе работы (например, добавлять драйверы устройств). Эта проблема решается путем реализации ***подключаемых модулей ядра*** *— программных компонентов ядра, динамически компонуемых (связываемых, линкуемых — link) средствами самого ядра без его перезапуска (перезагрузки).*

Более существенным недостатком монолитной архитектуры является то обстоятельство, что сбой в любом компоненте ядра (включая драйверы устройств или модули, предложенные сторонними производителями ПО) с большой вероятностью приведет к сбою всей ОС, так как само ядро как программа окажется в неконсистентном состоянии. Такие сбои крайне сложно изолировать, часто требуется перезагрузка ОС (компьютера). В то же время выгрузить отдельный компонент микроядерной архитектуры не составляет проблемы. Условно число ошибок в программном коде считается пропорциональным размеру кода, поэтому вероятность сбоя в пространстве ядра тем ниже, чем меньший код исполняется в пространстве ядра.

Наконец, любой компонент ядра ОС (в т. ч. модуль) монолитного ядра, драйвер оборудования и др. имеет полный контроль над ОС и компьютером, т. к. исполняется в пространстве ядра. Это может служить потенциальным источником проблем с безопасностью из-за ошибочных или злонамеренных действий производителей таких компонентов, что является еще одним аргументом в пользу сокращения количества кода, исполняемого в пространстве ядра.

Большинство используемых в настоящее время ОС базируются на монолитной архитектуре ядра (в частности, ядро Linux), или близкой к ней гибридной архитектуре (ядро Windows NT).

Абстракции ОС на примере открытия файла

Рассмотрим на примере открытия файла в ОС на базе ядра Linux, как происходит работа различных компонентов и уровней абстракции ОС (исключая управляющие компоненты, детальное описание алгоритмов работы которых лежит за рамками данного пособия). Пусть прикладной программист написал следующую строку на языке Си (или аналогичную на другом языке высокого уровня):

```
FILE *in_file = fopen (path_to_file, "r");
```

Данный код был транслирован в двоичный исполняемый код, специфичный для выбранной ***платформы*** *— среды исполнения программ (аппаратной*

— архитектуры процессора и программной — интерфейса прикладного программирования ОС). Пользователь запустил программу, содержащую данную строку, запросил открытие файла через ее интерфейс, что привело к исполнению данного кода. В случае использования интерпретируемых языков (например, Python) соответствующий двоичных код генерируется после запуска программы во время ее исполнения, что не меняет ситуацию. Аналогично работа будет осуществляться и при использовании языков программирования, транслирующих исходный код в промежуточный платформо-независимый байт-код, который потом интерпретируется на целевой платформе (например, JAVA).

1. Двоичный код содержит в себе вызов функции стандартной библиотеки языка — части независящего от ОС интерфейса, но реализованной через интерфейсы программирования ОС. Параметры функции и адрес следующей инструкции — точки возврата из функции — помещаются в регистры процессора или стек потока выполнения приложения с соответствующим смещением указателя стека, там же сохраняется точка возврата из функции — адрес следующей за вызовом функции инструкции. В ОС на базе Linux реализация стандартной библиотеки Си находится в библиотеке The GNU C Library (glibc, файл `libc.so`).

2. Файловые операции с данными типа `FILE` обеспечивают ***буферизированный ввод/вывод*** *— для каждого файла создается буфер некоторого объема (в пользовательском пространстве). При записи данные накапливаются в этом буфере, прежде чем будут отправлены в ОС, а при чтении данные поступают в буфер, прежде чем будут переданы библиотекой приложению.* Это делается для сокращения числа обращений к ОС и системных вызовов. Например, если программа будет прочитывать данные порциями по несколько байтов, накладные расходы на системные вызовы будут составлять существенную долю времени ее работы, в то время как более оптимально будет прочитать сразу группу в несколько сотен или тысяч байтов.

   До начала попытки открытия файла `fopen` проверяет корректность строки, переданной в качестве режима открытия. Если там указано что-либо недопустимое (например, `"t"` вместо `"r"`), функция возвращает `NULL` и устанавливает значение глобальной переменной стандартной библиотеки `errno` равным коду соответствующей ошибки — `EINVAL`.

Если все прошло успешно, fopen вызывает системную функцию open для открытия данного файла (O_RDONLY — флаг битовой маски для открытия файла только для чтения):

```
int fd = open (path_to_file, O_RDONLY);
```

3. Реализация функции open также находится в библиотеке glibc. Это оберточная функция для системного вызова, ее вызов — передача аргументов, точки возврата и возвращаемого значения, осуществляется обычным образом. Данная группа функций обеспечивает ввод и вывод непосредственно средствами ОС, без буферизации в пользовательском пространстве. Функция open делает сам системный вызов:

```
int ret = syscall(__NR_open, path_to_file,
                                    O_RDONLY, 0);
```

   Здесь __NR_open — номер системного вызова open, 0 — режим доступа к создаваемому файлу (не используется при открытии).

4. Функция syscall выполняет машинный код системного вызова (зависимый от архитектуры процессора). Обычно номер системного вызова помещается в выделенный для этого регистр, остальные параметры также записываются в регистры (в отличие от вызова функции передача значений параметров через стек обычно не производится; указатели, в данном случае path_to_file, также передаются через один из регистров как целые числа). После этого вызывается машинная инструкция системного вызова: происходит переключение в пространство ядра.

5. Получает управление специальная функция ядра, ответственная за системные вызовы. Эта функция также специфична для архитектуры процессора (например, для RISC-V это handle_syscall). Исполняются машинные инструкции по извлечению номера системного вызова из регистра и по *таблице системных вызовов* (структуре данных ядра, содержимое которой также специфично для архитектуры процессора) определяется адрес функции ядра sys_open, ответственной за данный системный вызов. При отсутствии системного вызова с таким номером возвращается соответствующий код ошибки. В данном случае, т. к. вызов пришел из функции-обертки, такая ошибка возникнуть не может иначе как в следствии некорректной сборки ядра или библиотеки.

   Не вдаваясь в подробности внутренней структуры ядра, отметим, что данные системные вызовы, отвечающие за файловые операции

(sys_open, sys_read, sys_write, sys_close и другие), обеспечивают *интерфейс абстракции над конкретными файловыми системами накопителей данных* — ***виртуальной файловой системы*** (***Virtual File System***, VFS). Это унифицированный программный интерфейс ОС для доступа к файлам, расположенным на разных типах накопителей данных и разных файловых системах. Пространство имен файлов в VFS представляет собой агломерацию полных путей к файлам, доступных из приложений, в то время как ***точка монтирования*** (***mountpoint***) в VFS *определяет конкретную файловую систему, ответственную за данный каталог.*

Функция sys_open может вызывать другие функции ядра, как обычные функции, без переключения уровня привилегий — но сами вызовы и их работа будут происходить в пространстве ядра. Для обеспечения безопасности и изоляции поток выполнения в пространстве ядра имеет отдельный, отличный от пользовательского пространства, стек.

6. Функция sys_open (точнее, вызываемые из нее функции ядра) копирует имя файла, расположенное по переданному виртуальному адресу в пользовательском пространстве процесса, в адресное пространство ядра. Это необходимое действие в условиях изоляции пространства ядра и пользовательского пространства.

   По таблице точек монтирования ядро определяет, *какой именно* ***драйвер файловой системы*** *и какое устройство* (не обязательно накопитель данных или его раздел — это может быть файл-образ раздела или сетевой каталог) *отвечает за запрашиваемый файл и какой у него путь относительно корневого каталога данной файловой системы*. Далее вызываются специфичные для драйвера файловой системы функции определения прав доступа к данному файлу. При отсутствии запрашиваемого файла возвращается ошибка `-EEXIST`. По контексту процесса, сделавшему системный вызов, определяется владелец и группа владельца процесса, сверяются права доступа к файлу (и всей ветке дерева каталогов до него). При отсутствии соответствующего доступа к каталогу или файлу возвращается соответствующая ошибка — значение `-EACCES` (то есть код ошибки со знаком минус), в противном случае вызывается соответствующая функция открытия файла данной файловой системы.

7. Функции драйвера файловой системы проводят необходимые операции ***со слоем абстракции соответствующего типа накопителей***

информации *(например,* ***подсистемы блочного ввода/вывода*** *или* ***сетевой подсистемой****) для получения и записи данных.* Функции подсистемы обращаются к драйверам конкретных накопителей (например, блочных устройств) и аппаратных интерфейсов для непосредственного осуществления операций ввода/вывода. При обнаружении ошибок на соответствующем этапе возвращаются их коды.

8. Получив информацию об успешном открытии файла, функция sys_call создает в ядре соответствующий открытому файлу дескриптор, записываемый в таблицу открытых файлов данного процесса. Номер этого дескриптора (положительный, в том числе нулевой) или код ошибки (отрицательный), возвращается как результат системного вызова через предназначенный для этого регистр. Управление передается в пользовательское пространство, обратно в функцию open.
9. Функция open проверяет знак возвращенного значения `ret` и возвращает файловый дескриптор в случае успеха, выставив значение errno в `0`, или выставляет значение errno в `-ret`, и возвращает отрицательное значение. Управление возвращается в функцию `fopen`.
10. В случае ошибки функция fopen возвращает `NULL`. При успешном открытии файла ей нужно выделить память под буфер пользовательского пространства и под структуру `FILE`, которая как минимум должна содержать указатель на данный буфер и файловый дескриптор, предоставленный ОС. Это может быть сделано с помощью функции malloc. Не вдаваясь в подробности, скажем, что данная функция может выделять блоки памяти из ранее полученных от ОС блоков большего размера или запрашивать блоки с помощью соответствующего системного вызова, расширяющего виртуальное адресное пространство процесса. Если malloc вернет ошибку (`NULL`), то fopen в свою очередь выставит код ошибки `ENOMEM` в переменную errno и возвращает значение `NULL` (при такой ошибке файл не должен открываться с помощью open вообще или быть закрыт посредством функции close, если был открыт ранее). Эта память должна освобождаться при вызове функции fclose. Функция fopen возвращает указатель на созданную структуру `FILE` — через стек или регистр — после чего передает управление по сохраненному адресу точки возврата.

Хотя эта процедура описана несколько упрощенно, она отражает реальную ситуацию. Кроме описанных здесь моментов, существует также буферизация и кеширование данных и метаданных (прав доступа и т. д.) файлов

внутри ядра ОС и драйверов устройств, также оптимизирующая количество операций ввода/вывода.

Помимо этого операция ввода/вывода (на шаге 7) может занять длительное время, как собственно из-за работы устройства, так и, например, из-за ожидания очереди ввода/вывода данных, поступающих от различных запущенных приложений. Кроме того, при вводе данных может потребоваться ожидание их поступления, например, от пользователя (нажатие на клавишу) или по сети. При записи данных может оказаться, что системный буфер записи заполнен и нужно ждать, когда устройство выведет эти данные. В таких случаях, если пользовательское приложение не предприняло дополнительных шагов для возможности продолжения выполнение других действий в ожидании ввода/вывода данных, возврат из системного вызова не должен производиться, пока операция ввода/вывода не будет завершена.

Поток, который не может продолжить выполнение, пока не наступит какое-то внешнее по отношению к данному потоку событие, называется ***заблокированным***. Системный вызов, исполнение которого может привести к блокировке потока, называется ***блокирующим***. В общем случае заблокировать исполнение могут не только системные вызовы, связанные с исполнением операций ввода/вывода, но и другие. Это и вызовы, напрямую связанные с ожиданием какого-то события — приостановка по таймеру (sleep), ожидание завершения дочернего процесса (wait), и любые вызовы, для выполнения которых ядру требуется получить ***монопольный доступ*** к какому-то ***общему ресурсу*** (внешнему устройству или структуре данных ядра) — в этом случае нужно ждать, пока другой поток (параллельно на другом процессоре или будучи вытесненным при логическом параллелизме), выполняющий свой системный вызов и уже завладевший данным ресурсом, завершит свою работу.

Поскольку заблокированный поток не исполняется и не претендует на процессорное время, операционная система во многих случаях может приостановить его исполнение — ***перевести в спящее состояние*** — *и передать управление другому, готовому к исполнению, потоку.*

Управление потоками и процессами

Для управления процессами и потоками выполнения ядро ОС ведет ***таблицы потоков и процессов***. В реализации ядра Linux используется единая таблица задач, хотя различные ОС могут использовать для этих целей отдельные таблицы, например, вести таблицу потоков каждого процесса. В системах, поддерживающих только однопоточные процессы, управление потоками сводится к управлению процессами.

К элементам и контексту, ассоциированному с процессом, относятся:

- ***учетные и регистрационные данные процесса***, такие как идентификатор, имя, владелец, группа владельца и другие;
- ***виртуальное адресное пространство***, содержащее код процесса, глобальные данные процесса, данные, выделенные из кучи — общие для всех потоков, а также стек и глобальные данные каждого потока этого процесса и соответствующие ***привилегированные регистры процессора и структуры данных в памяти***, ответственные за организацию виртуального адресного пространства и трансляцию адреса;
- ***выделенные ресурсы и ассоциированные данные в ОС***, например, открытые файлы, зарегистрированные обработчики сигналов, общая для нескольких процессов память, переменные окружения и другие элементы;
- ***потоки выполнения*** этого процесса;
- ***иерархия наследования процессов***, т. е. родительский по отношению к данному процесс и его дочерние процессы.

К элементам и контексту, ассоциированному с потоком, относятся:

- ***собственный стек*** — хотя все потоки процесса исполняются в одном адресном пространстве и могут иметь доступ ко всем данным всех потоков того же процесса, в т. ч. имеют общий сегмент кода (и часто исполняют один и тот же код), создание собственного стека для каждого потока является необходимым для реализации возможности вызова разными потоками разных функций и хранения их локальных данных;
- ***регистры процесса пользовательского пространства***, обеспечивающие непосредственное исполнение кода, включая ***указатель инструкции*** и ***указатель стека***;
- ***стек ядра***, в котором хранятся локальные данные обработчиков системных вызовов, делаемых данным потоком.

Заметим также, что ядро также может обеспечивать отдельный стек для обработчиков аппаратных прерываний, чтобы их данные не конфликтовали с данными обработчиков системных вызовов.

Каждый зарегистрированный в системе процесс (в однопоточном случае) или каждый поток выполнения может находиться в одном из нескольких состояний:

- ***исполняющийся*** *(сейчас использует процессорное время),*

- ***готовый (к исполнению —*** *ожидает процессорное время, но вытеснен другим потоком или процессом),*
- ***спящий (заблокированный*** *блокирующим системным вызовом из-за невозможности продолжения исполнения до наступления внешнего события — в ожидании ввода/вывода, тайм-аута, освобождении общего ресурса ядра, завершения дочернего процесса или другого — и не претендующий на процессорное время),*
- ***создаваемый*** *(процедура создания процесса или потока еще не завершена, но он уже зарегистрирован в ядре),*
- ***завершаемый*** *(процедура завершения потока или процесса уже начата, но еще не завершена — например, освобождаются ресурсы, задействованные процессом, закрываются открытые файлы),*
- ***зомби*** *(исполнение процесса завершено, ресурсы освобождены, но его код возврата —* ***exit status*** *— не прочитан родительским процессом: необходимо хранить запись о нем для того, чтобы прочтение кода возврата было возможным),*
- *и других — точный перечень состояний зависит от того, как управление процессами и потоками реализовано в ОС.*

Одной из управленческих задач ОС является ***планирование потоков*** *— выстраивание порядка исполнения потоков.* За выбор момента переключения контекста и следующего исполняемого потока отвечает ***алгоритм планирования***. Если текущий поток приостанавливается сделанным им блокирующим системным вызовом, то он переводится в состояние «спящий», а планировщик с помощью алгоритма планирования выбирает один из готовых к исполнению потоков того же или другого процесса и производит соответствующее переключение контекста. (На самом деле не всегда есть возможность перевести заблокированный поток в спящее состояние, он может остаться работающим — исполняющимся или готовым к исполнению — тонкости реализации этого описаны на стр. 50). Также алгоритм может принять решение о переключении контекста по прерыванию от таймера, тогда исполняющийся поток переводится в состояние готового, а планировщик выбирает для переключения и исполнения другой готовый поток. Также потоки могут сами, с помощью специального системного вызова, освобождать процессорное время для других процессов, активизируя планировщик, который переведет их в состояние готового к исполнению.

Парадигма проектирования ОС, где приложения сами освобождают процессор, называется ***кооперативной многозадачностью****, а парадигма,*

*при которой ОС принимает решения о переключении контекста — **вытесняющей (премптитивной) многозадачностью***. Чистая кооперативная многозадачность возлагает на прикладного программиста ответственность за освобождение процессора, что сопряжено с очевидными рисками для ориентированных на пользователей ОС (***интерактивных**, где основная задача — реакция на внешние запросы*). Более того, на таких системах переключение контекста должно быть достаточно частым (каждые несколько десятков миллисекунд), чтобы обеспечить приемлемое для пользователя время отклика от всех запущенных им приложений. В то же время на *системах, направленных на большие вычисления (**пакетных**)*, например, на суперкомпьютерах, трата ресурсов на частое переключение контекста является избыточной, поэтому более оптимальной является невытесняющая стратегия, когда поток выполняется до тех пор, пока не будет завершен или переведен в спящее состояние.

При невытесняющей стратегии может использоваться алгоритм планирования простой очереди, в то время как в интерактивных системах простейшим вариантом является ***циклическое переключение*** по таймеру (алгоритм «***round-robin***»), когда всем потокам выделяется равный квант времени (квант должен быть не слишком маленьким, чтобы не тратить критичную долю времени на переключения, и не слишком большим, чтобы не нарушить интерактивность). Более сложные варианты включают себя использование ***приоритетного планирования***, выделяющего квант времени, пропорциональный приоритету, ***совершенно справедливого планирования***, выделявшего каждому процессу пропорциональную долю процессорного времени с учетом приоритета (используется в ядре Linux) и другие.

Межпроцессное взаимодействие

Поскольку процессы работают в отдельном виртуальном адресном пространстве, они не могут просто обратиться к одному и тому же адресу в памяти для обмена информацией. Для решения задачи обмена данными между процессами операционные системы предоставляют специальные ***механизмы межпроцессного взаимодействия*** посредством соответствующих системных вызовов. Приведем наиболее известные из них.

1. ***Общий файл*** — простейший (не требующий отдельной реализации со стороны ОС), но не всегда эффективный и удобный в применении способ обмена данными между процессами.
2. ***Канал (pipe)*** *— файл особого рода, в который можно записывать с одного конца и прочитывать с другого.* Различают ***анонимные каналы***, доступные процессу и его потомкам по общему файловому де-

скриптору, а также ***именованные каналы*** — файлы VFS особого рода, которые могут открывать независимые процессы по его имени. Канал представляет собой последовательность передаваемых байтов (очередь данных, иногда обозначается аббревиатурой `FIFO` — ***first in, first out***, первым пришел — первым обслужен). В отличие от файла, у канала нет произвольного доступа к данным: ранее полученные (прочитанные) данные не сохраняются, имеется только последовательная запись и последовательное чтение ранее записанных данных. В остальном (кроме этапа создания) работа с ними проводится через обычные файловые операции — в программе канал представлен в виде двух файловых дескрипторов: дескриптор для записи и дескриптор для чтения.

При чтении из пустого канала читающий процесс (***потребитель***, ***consumer***) должен быть заблокирован в ожидании данных, т. е. до тех пор, пока записывающий процесс (***производитель***, ***producer***) не запишет их. На уровне ядра канал может быть реализован, например, посредством ограниченного кольцевого буфера (очереди). В этом случае при попытке записи в полный буфер производитель также будет заблокирован в ожидании момента, когда потребитель прочитает часть данных.

Стандартные потоки (***stream***) ввода, вывода и ошибок, открытые для каждого приложения, на самом деле могут являться каналами. Например, терминал записывает вводимые с клавиатуры данные в дескриптор записи канала потока стандартного ввода, а поток выполнения прочитывает их из дескриптора чтения. Для потоков вывода и ошибок ситуация обратная.

3. ***Общая память и отображение файла в память (анонимное или именованное)*** *— позволяет отобразить содержимое общего файла в виде блока памяти, выделенной из кучи.* Работа с ним производится обычными инструкциями работы с памятью.
4. ***Сигналы*** *— короткие сообщения, посылаемые процессу операционной системой или другим процессом*. Сообщение представляет собой номер сигнала. При получении сигнала ОС прерывает исполнение процесса и передает управление специальной функции — обработчику сигнала, предоставленному процессом или стандартному, предоставленному ОС. При поступлении сигнала процесс может выполнить некоторую команду, быть приостановлен, продолжить работу, завершен «мягко» (средствами самого процесса) или принуди-

тельно (средствами ОС). Также сигнал может быть сгенерирован самой ОС при возникновении некоторых ошибок и исключений на уровне процессора (некорректная инструкция, деление на ноль и т. п.). По принципу реализации механизм сигналов напоминает механизм прерывания, но в рамках пользовательского приложения: каждому номеру сигнала сопоставляется обработчик.

5. Другие механизмы, например, ***сетевые сокеты*** (***socket***), позволяющие процессам, запущенным на одном или разных компьютерах, обмениваться информацией по компьютерной сети; ***очереди сообщений***, ***удаленный вызов процедур (Remote Procedure Call***, RPC***)*** и ***интерфейс передачи сообщений (Message Passing Interface***, MPI***)***. Среди них могут быть как реализованные средствами ОС, так и на уровне библиотек пользовательского пространства поверх системных механизмов. Описание этих механизмов лежит за рамками данного пособия.

Для обмена информацией между потоками тоже могут использоваться некоторые из указанных механизмов, например, каналы или аналоги сигналов. Однако, т. к. потоки одного процесса работают в одном адресном пространстве, отдельной задачи обеспечения передачи данных от одного потока к другому потоку того же процесса перед ОС не стоит.

Синхронизация потоков выполнения

В параллельном программировании нередко возникает задача синхронизации потоков выполнения. К таким ситуациям относятся:

- приостановка одного из потоков в ожидании данных, поступающих от другого потока (***задача*** организации взаимодействия ***производителя и потребителя***, в т. ч. ожидание данных с устройства ввода/вывода);
- ***состояние гонки*** при обращении к общему ресурсу (например, к общей переменной, к общему аппаратному устройству, работа с которым требует монопольного доступа).

На уровне ОС также требуется синхронизация: ядро следует рассматривать как многопоточный процесс, потому что потоки пользовательских процессов могут делать параллельные запросы к ядру. Это относится в том числе к однопроцессорной многозадачной системе, обеспечивающей логический параллелизм: во время исполнения кода системного вызова может произойти переключение контекста на другой поток или процесс, который может сделать тот же или другой системный вызов, т. е. начать работать с теми же об-

щими данными ядра или аппаратными устройствами. Кроме того, работа кода ядра может быть прервана асинхронным аппаратным прерыванием, обработчик которого может обращаться к общим ресурсам ядра.

Монопольный доступ может потребоваться, например, для многих устройств ввода/вывода. Так, если принтер получит данные для печати от двух работающих приложений, пытающихся одновременно распечатать два документа, не последовательно, а «вперемешку», то результат печати будет непредсказуем. Управление устройствами через взаимодействие с контроллером также требует осуществления этой манипуляции одним приложением во избежании получения противоречивых команд и приведение устройства в неконсистентное состояние. В таких случаях ресурс (обычно аппаратный) иногда может быть просто выделен в управление одному компоненту ОС — обработчику аппаратного прерывания, который на время обработки запрещает (***маскирует***) прерывания от того же устройства, тем самым исключая конфликт.

Общие ресурсы, например, программные ресурсы ОС (файлы, каналы и др.), и иные общие данные ядра — переменные и структуры данных — могут разделяться между потоками (обработчиками системных вызовов) и компонентами ядра. В этом случае в момент обращения к такому ресурсу требуется монополизация доступа к нему. Например, одновременный несогласованный доступ к общей структуре данных (массиву или дереву поиска) может привести к получению некорректных данных или вообще приведению всей структуры в нерабочее состояние.

Более того, даже такая простая инструкция языка программирования, как инкремент переменной, не является ***атомарной (atomic)*** — для ее исполнения требуется прочитать значение переменной в регистр, увеличить его на единицу и записать в память новое значение. Так это может выглядеть на языке псевдоассемблера:

```
LOAD  REG VAR # загрузить переменную VAR в регистр REG
ADD   REG 1   # добавить 1 к значению в регистре REG
STORE REG VAR # сохранить значение из регистра REG в VAR
```

Если два потока в результате переключения или одновременной работы выполнят эти действия в таком порядке

```
# Изначально: VAR = 3
LOAD  REG VAR  # Поток 1, его REG = 3, общий VAR = 3
ADD   REG 1    # Поток 1, его REG = 4, общий VAR = 3
LOAD  REG VAR  # Поток 2, его REG = 3, общий VAR = 3
STORE REG VAR  # Поток 1, его REG = 4, общий VAR = 4
```

```
ADD   REG 1    # Поток 2, его REG = 4, общий VAR = 4
STORE REG VAR  # Поток 2, его REG = 4, общий VAR = 4
```

то в переменную VAR записывается значение 4, вместо требуемого значения 5. Реальный порядок выполнения непредсказуем и результат может оказаться как корректным, так и некорректным, в зависимости от условий запуска.

Абстрактно синхронизация при состоянии гонки разрешается при помощи механизмов ***взаимного исключения*** *выполнения некоторого действия двумя и более потоками.* Для этого общий ресурс снабжается ***блокировкой (блокировщиком, замком, lock)*** *— специальной переменной, которая может находиться в двух состояниях —* ***захваченном*** *или* ***свободном*** *и, соответственно, поддерживает две операции —* ***захвата (acquire)*** *и* ***освобождения (release) блокировки***. Операции реализуются таким образом, что поток A, пытающийся захватить блокировку, проверяет, не захвачена ли она уже другим потоком. Если нет — блокировка захватывается потоком A и его исполнение потока продолжается. Если блокировка уже захвачена другим потоком B, то поток A ***блокируется*** в ожидании пока поток B не выполнит освобождение блокировки, после чего поток A захватывает блокировку и продолжает исполнение.

Таким образом, организация взаимного исключения доступа к общему ресурсу состоит в том, что каждому такому ресурсу сопоставляется его блокировка, а каждый поток, который хочет воспользоваться этим ресурсом, должен предварительно захватить блокировку — захватить ресурс в монопольное использование, — а по завершении операции освободить ее (блокировку) — освободить ресурс. Сразу обратим внимание на несколько обстоятельств.

- Блокировки логически ***атомизируют*** набор операций языка высокого уровня или машинных инструкций, производимых между захватом и освобождением (по отношению к такому участку кода, параллельное исполнение которого не допускается, может также использоваться термин ***критическая секция***).
- *При работе с общими ресурсами обязательно нужно захватывать защищающие их блокировки* «вручную», средства языков программирования не предусматривают полную автоматизацию этого действия (кроме как посредством использования соответствующих функций-оберток для доступа к общему ресурсу). В этом смысле использование блокировки является *факультативным — но отказ от ее захвата чаще всего будет являться ошибкой программирования. Также критично важно освобождать захваченные блокировки* по заверше-

нии работы с ресурсом, так как в противном случае другой поток или даже тот же поток при следующем обращении к этому ресурсу рискует оказаться в вечном ожидании захвата блокировки (состоянии вечной блокировки).

- Блокировки нужно захватывать на как можно меньшее время, чтобы минимизировать простои потоков в заблокированном состоянии — т. е. в ожидании их (блокировок) освобождения: особенно это касается блокировок, требующих активного ожидания, описанного ниже;

Как уже было отмечено, в ядре операционной системы также возможно возникновение состояния гонки, например, из-за того, что два и более потока одновременно сделают системный вызов — как при одновременной многопоточности на разных ядрах процессора, так и при логическом параллелизме в рамках одного ядра процессора. Таким образом, от ОС требуется обеспечить механизмы синхронизации потоков выполнения внутри ядра. В ОС используется два основных (базовых, примитивных) механизма синхронизации: ***спин-блокировки*** (***spinlock***, ***циклические блокировки***) и ***блокировки с приостановкой***.

Спин-блокировки просты в реализации с помощью специальной ***атомарной машинной инструкции***, которая одновременно проверяет и устанавливает в 1 значение выбранной переменной-замка. Если старое значение было 0, то поток захватывает блокировку, если 1, то блокировка была захвачена другим потоком и нужно делать циклическую проверку значения переменной-замка, пока оно не станет равным 0, после чего можно будет захватить блокировку. Недостатком такого механизма является расход процессорного времени во время ожидания захвата блокировки (***активное ожидание***) без выполнения полезной работы. Такой механизм не годится для длительного ожидания, но подходит, например, для реализации ***атомарных переменных***, т. е. переменных, арифметические операции с которыми выполняются атомарным образом, без возможности вмешательства со стороны других потоков. Также этот механизм может быть необходим для реализации блокировок с приостановкой.

Специальные атомарные инструкции требуются потому, что «обычные» инструкции не гарантируют атомарность выполнения операции даже на одном ядре, могут быть слишком просты, чтобы выполнить одновременно чтение и проверку, не защищают от одновременного вмешательства от других процессоров (ядер). Реализация блокировок без аппаратной поддержки (чисто алгоритмически) возможна, но трудоемка — требует накладных расходов при работе. С другой стороны, атомарные инструкции, поддерживаемые боль-

шинством современных архитектур процессоров, позволяют даже организовать атомарные переменные без использования блокировок, что ускоряет работу системы.

Блокировки с приостановкой решают проблему активного ожидания: поток, которому нужно ждать освобождения блокировки, переводится в спящее состояние. Спящий поток не использует процессорное время, которое отдается другому потоку. Освобождение блокировки с приостановкой вызывает активацию («пробуждение») других спящих в ожидании захвата блокировки потоков. Это оптимальный вариант для пользовательского пространства и многих задач пространства ядра. Однако он *неприменим в некоторых местах, в частности, в обработчиках аппаратных прерываний, так как приостановка в них недопустима*. Приостановка в обработчике аппаратного прерывания чревата рядом проблем, в т. ч. потому, что в этот момент могут быть запрещены другие прерывания, а длительный их запрет приведет к нестабильной работе системы.

В библиотеках параллельного программирования и ядре Linux реализованы различные виды блокирокок с приостановкой — ***семафоры (semaphore)***, ***мьютексы (mutex)*** и др., обладающие разными возможностями. Мьютексы являются наиболее простыми в применении, поскольку, как и спин-блокировки, имеют только два состояния — «разблокирован» или «заблокирован».

Реализация блокировок с приостановкой требует использования атомарных переменных, реализуемых аппаратно или посредством спин-блокировок. Заметим, что допускается вложенный захват спин-блокировок и блокировок с приостановкой, а также захват спин-блокировок при уже захваченных блокировках с приостановкой, но в большинстве случаев запрещается захват блокировок с приостановкой при захваченной спин-блокировке, т. к. это может привести к длительному активному ожиданию захвата блокировки, удерживаемой другим, приостановленным, потоком. Также, на однопроцессорной системе, если во время попыток захвата спин-блокировки запрещены прерывания и переключение контекста, а другой поток, удерживающий спин-блокировку, оказывается вытеснен и не может получить процессорное время, то система приходит в состояние бесконечного ожидания.

Вложенные захваты блокировок могут понадобиться если, например, потоку требуется обращение к новому общему ресурсу (переменной, структуре данных), когда уже используется другой общий ресурс. В этом случае может возникнуть ситуация ***взаимоблокировки***, *когда образуется группа из двух и более потоков, каждый из которых и удерживает ресурсы, и ожидает выде-*

ления ресурсов, удерживаемых потоками той же группы. Например, поток A удерживает ресурс 1 и заблокирован в ожидании ресурса 2, а поток B удерживает ресурс 2 и заблокирован в ожидании ресурса 1. В этой ситуации работа потоков не может быть продолжена, они будут находиться в «вечном» ожидании друг друга. Взаимоблокировки труднообнаружимы и практически непредотвратимы программными средствами, а при их обнаружении практически никогда не могут быть устранены универсальным образом. Если в пользовательском пространстве решение о прерывании «зависшего» потока (процесса, приложения) может принять пользователь, то взаимоблокировка в пространстве ядра может привести к полной остановке ОС.

Для недопущения взаимоблокировок следует всегда захватывать ресурсы (т. е. соответствующие им блокировки) строго в определенном порядке. Например, если поток A сначала захватывает ресурс 1, а затем — ресурс 2, то и поток B должен захватывать сначала ресурс 1, потом ресурс 2, а освобождение ресурсов каждый из них должен проводить в обратном порядке. При таком подходе взаимоблокировка возникнуть не может. При проектировании компонентов ядра ОС нужно всегда внимательно следить за порядком осуществления вложенных захватов блокировок.

Отметим, что «блокировки» могут также называться «***замками***», захват блокировки — ***установкой блокировки***, ***блокированием ресурса*** (от доступа со стороны других потоков), ***захватом ресурса*** (в монопольное использование) и т. п., освобождение блокировки — снятием блокировки, ***разблокированием ресурса***, ***освобождением ресурса*** и т. д.

Необходимость перевода потока в спящее состояние может возникнуть не только при доступе к общим ресурсам, но и при ожидании данных при вводе, ожидании освобождения буфера при записи (в задаче производителя и потребителя), при ожидании завершения другого потока или процесса и т. п. Блокирующий системный вызов — за исключением случая блокирования в ожидании захвата спин-блокировки — приведет именно к тому, что поток будет переведен в спящее состояние. В этом случае также могут произойти взаимоблокировки, например, если поток процесса A заблокирован в ожидании данных, посылаемых потоком процесса B через канал, но этот же поток процесса B ожидает завершения процесса A, то потоки A и B окажутся в состоянии «вечной» блокировки.

Описанные здесь проблемы и приемы синхронизации относятся как к потокам одного многопоточного процесса (ядру в целом или параллельному приложению пользовательского пространства), так и к взаимодействующим потокам разных пользовательских процессов, в т. ч. однопоточных. В послед-

нем случае синхронизация может описываться на языке синхронизации самих процессов в контексте доступа к общей памяти (если таковая реализована и поддерживается ядром) или, как в примере выше, ожидании завершения процесса A процессом B, а процессом B — поступления данных по каналу от процесса A: имеется в виду, что взаимодействуют и синхронизируются единственные потоки соответствующих процессов. Таким образом, на уровне и прикладного, и системного программирования следует грамотно строить взаимодействие потоков и процессов при совместной работе.

Управление памятью

Механизмом трансляции адреса, наиболее полно отвечающим потребностям виртуальной памяти ОС, и используемым в современных процессорах является ***страничная адресация памяти***. При таком способе адресации памяти вся физическая память подразделяется на страницы фиксированного размера (например, 4 килобайта). Специальный защищенный от изменения в пользовательском режиме регистр указывает на ***таблицу страниц***. Таблица страниц содержит дескрипторы страниц — записи, включающие в себя физический адрес начала страницы и атрибуты. Виртуальный адрес содержит в себе номер страницы в таблице и смещение внутри страницы. Атрибуты страницы могут включать в себя, например, права доступа к странице (чтение, запись, выполнение), показывать, был ли доступ или модификация страницы и другие параметры. Страничный подход имеет ряд преимуществ.

- Непрерывный в виртуальном адресном пространстве блок данных не обязан быть таким в физическом пространстве. Переполнение значения смещения внутри страницы в виртуальном адресе (т. е. переход к следующему байту после последнего на данной странице) приводит к переходу к следующему индексу в таблице страниц. Это позволяет избежать проблемы фрагментации физической памяти, когда новый непрерывный блок нужного размера не может быть выделен, хотя свободной памяти достаточно.
- Страницы могут быть выгружены в долговременную память, тогда при обращении к ним процессор сгенерирует исключительную ситуацию — ***ошибку отсутствия страницы***. ОС может обработать ее, подгрузив нужную страницу. Это происходит прозрачно для процесса. Также ОС может прозрачно изменить физический адрес страницы без изменения виртуальных адресов процесса.
- При создании ***клона*** процесса не обязательно копировать все данные текущего процесса до тех пор, пока они не будет измены. Например,

системный вызов fork в Linux создает дочерний процесс как клон текущего процесса, то есть код и данные (включая стек) созданного процесса будут идентичны родительскому процессу. В дальнейшем эти два процесса могут продолжить параллельную работу или один из них может быть замещен другой программой с помощью вызова функций группы exec. В первом случае далеко не все данные могут быть изменены различным образом в дочернем и в родительском процессе, а во втором данные клона вообще окажутся невостребованы. Поэтому полное копирование данных в памяти при создании клона является нецелесообразным, вместо этого можно оставить одну физическую копию данных в памяти, защитив страницу от записи.

Таблицы страниц каждого процесса будут ссылаться на одну и ту же физическую страницу. При попытке записи в нее процессор сгенерирует исключительную ситуацию — ***ошибку нарушения защиты страницы***. ОС может обработать эту ситуацию путем создания копии физической страницы и соответствующей правки таблицы страниц процессов, после чего процессы смогут продолжить работу. Такая технология называется ***копированием при записи*** (***Copy-on-Write***, ***CoW***).

- С помощью отображения виртуальных адресов разных процессов на одну и ту же защищенную от записи физическую страницу можно также экономить физическую память, не держа в ней несколько копий кода программы и разделяемых библиотек.

При нехватке физической памяти (например, при запросе процессом выделения новой памяти или необходимости подгрузить ранее выгруженную в долговременную память страницу) ОС должна выбрать страницу, которую следует выгрузить (вытеснить) из оперативной памяти в долговременную, чтобы заместить ее новой или подгружаемой страницей. За это отвечает ***алгоритм замещения страниц***. Фактически такой алгоритм является алгоритмом ***кеширования***, где физическая (быстрая) память рассматривается как кеш (медленной) виртуальной памяти. Часто наиболее эффективным алгоритмом кеширования является алгоритм вытеснения страницы, которая не использовалась дольше всего (LRU). Однако применение такого алгоритма в качестве алгоритма замещения страниц памяти невозможно, так как операционная система не имеет возможности контролировать каждое использование страницы, т. е. каждое обращение процесса к ней (в отличие, например, от обращений к данным на накопителях информации, осуществляемых с помощью системного вызова, а не автоматически процессором).

Обычно страницы снабжаются битами доступа и модификации, записываемыми в качестве атрибутов в таблице страниц. Каждый раз при чтении или изменении данных страницы процессор выставляет соответствующий бит в 1. ОС может периодически (например, по таймеру) занулять данные биты, тем самым определяя было ли чтение или модификация данной страницы с момента последнего зануления. Это позволяет ранжировать страницы по времени обращения, хотя и с точностью до интервала проверки, а не каждого обращения к странице, приблизив алгоритм замещения страниц к LRU.

Современные ОС на базе ядра Linux часто настраиваются таким образом, чтобы выгрузка производилась не в долговременную память, а на специальное виртуальное устройство — сжатый виртуальный накопитель. Данные этого накопителя располагаются в оперативной памяти, но сохраняются в сжатом виде. Таким образом, данные процессов остаются в оперативной памяти, что экономит время и ресурс твердотельных накопителей по сравнению с выгрузкой на долговременный накопитель. Оперативная память в свою очередь экономится за счет сжатия данных — практика показывает, что страницы памяти сжимаются довольно хорошо (в несколько раз).

Общая схема обработки ловушек и переключения контекста

Опишем общую аппаратно-программную схему обработки ловушек (ошибок, исключений, системных вызовов и аппаратных прерываний), а также переключения контекста потока и процесса, которое может произойти при их обработке и только при их обработке. Здесь описывается только концептуальная схема, детали и тонкости реализации зависят от конкретной архитектуры и ОС.

1. Предварительно ОС загружает код обработчиков аппаратных прерываний и других ловушек в память (данный код является частью ОС или, для аппаратных прерываний, может являться частью драйвера устройства). Это может происходить на стадии загрузки или при подключении устройства. ОС настраивает процессор привилегированными инструкциями — записывает в нужные места адреса точек входа в ОС при наступлении ловушки, задает целевой уровень привилегий (обычно это режим ядра ЦПУ), разрешает или запрещает те или иные прерывания.
2. Пусть в какой-то момент времени процессор исполняет поток A и наступило одно из следующих событий:

- поток программно запрашивает прерывание/системный вызов специальной инструкцией (предварительно записывает номер системного вызова в один из регистров, в другие регистры или в стек записываются входные параметры системного вызова);
- поток исполняет инструкцию, вызывающую ошибку/исключение (например, целочисленное деление на 0, некорректный виртуальный адрес, некорректная инструкция и т. д.);
- устройство ввода/вывода или таймер инициирует аппаратное прерывание (поступили данные, завершена ранее запрошенная операция ввода/вывода, наступил тайм-аут и т. п.) и выставляет особый бит шины: контроллер прерываний идентифицирует устройство и посылает сигнал процессору.

После этого на процессоре срабатывает ловушка.

3. Процессор
   - Выставляет привилегированные регистры, содержащие информацию о причине прерывания (тип ловушки, номер IRQ, код ошибки и т. д.).
   - Находит адрес точки входа в код ОС, ответственный за обработку ловушки, передает управление по этому адресу (это может быть один обработчик на все случаи или адрес конкретного обработчика определяется в зависимости от номера прерывания или кода ошибки), повышает уровень привилегий. В результате исполнение потока оказывается прерванным, исполняется код ядра в режиме ядра. Непривилегированные регистры, обеспечивающие исполнение потока A, остались неизменными.
4. Операционная система сохраняет значения регистров (контекст) прерванного потока и его процесса, подгружает значения регистров процесса ядра (стек аппаратного прерывания или стек ядра потока, таблицу страниц ядра и т. д.). По типу ловушки (аппаратное или программное прерывание, ошибка или исключение) и коду события (номер IRQ, код ошибки, номер системного вызова из регистра) или по факту получения управления в определенный обработчик находит функцию, непосредственно ответственную за обработку данного системного вызова, данной ошибки/исключения, аппаратного прерывания от данного устройства и вызывает ее — как обычную функцию ядра.

При обработке аппаратного прерывания запрещаются (***маскируются***) аппаратные прерывания по крайней мере от того же устройства, которое его вызвало.

5. Процессор выполняет код обработчика. Обработчик выполняет необходимые действия (общается с устройством, вызвавшим прерывания — получает данные, настраивает дальнейшие операции ввода/вывода, время следующего прерывания; определяет тип ошибки/исключения и как на него реагировать; выполняет запрошенные системным вызовов действия и т. д.), возвращает результаты обработки ОС.
6. ОС принимает решения о дальнейших действиях:
   - если ошибка не фатальна для процесса потока A — устранена (например, данные подгружены в ОЗУ из виртуальной памяти или может быть обработана процессом (например, процесс зарегистрировал обработчик сигнала на случай деления на 0), аппаратное прерывание обработано (т. е. обрабатывалось асинхронное событие вне контекста пользовательского потока и его процесса), по решению планировщика потоков и процессов переключение по таймеру на другой поток не требуется — нужно продолжить исполнение прерванного потока A;
   - если поток A должен ждать результатов работы системного вызова, например, завершения запрошенной операции ввода/вывода, поступления данных, или код обработчика должен ждать освобождения спящей блокировки — т. е. поток A не может быть продолжен и должен освободить процессорное время (произведен блокирующий системный вызов) — то нужно заблокировать поток A с переводом в спящее состояние и переключиться на поток B того же или другого процесса;
   - если поток A может исполняться дальше, но по решению планировщика процессорное время требуется передать другому потоку, то нужно приостановить поток A (перевести в состояние «готовый») и переключиться на поток B того же или другого процесса;
   - если ошибка фатальна для процесса потока A (например, запрошенный виртуальный адрес не принадлежит адресному пространству процесса; процесс не зарегистрировал обработчик на случай деления на 0 и т. д.), то нужно принудительно завершить процесс потока A и переключиться на поток B другого процесса;

- если поток A сам запросил свое завершение или завершение всего своего процесса, выполнив специальный системный вызов (поток и процесс обязаны сделать такой системный вызов в конце своей работы, иначе процессор будет исполнять мусорные данные, находящиеся в памяти после кода процесса, что приведет к непредсказуемому поведению), то нужно штатно завершить поток A (и, возможно, весь его процесс) и переключиться на поток B того же (при наличии) или другого процесса.

7. ОС выполняет необходимые действия:
   - если в результате обработки ловушки другой, заблокированный на некотором системном вызове (спящий) поток C может продолжить работу (завершен ввод/вывод, освобождена блокировка, закончено ожидание какого-то события), то этот поток разблокируется (переводится в состояние «готовый»);
   - если работа процесса потока A завершается — освобождает его ресурсы (например, открытые файлы, выделенную из кучи память) и сохраняет информацию об ошибке/коде завершения (процесс переводится в состояние «зомби»);
   - Если требуется переключение контекста, планировщик выбирает поток B, на который нужно переключиться (это может быть и только что разблокированный поток C) и переключается на стек ядра потока B (загружает регистры пространства ядра потока B) — т. е. собственно переключает контекст;
   - если обрабатывалось аппаратное прерывание, сообщает контроллеру прерываний о завершении обработки аппаратного прерывания (***снимает прерывание***) — иначе это же прерывание от того же устройства тут же сработает вновь — и разрешает прерывания;
   - если обрабатывался системный вызов, сохраняет в соответствующем регистре возвращаемое значение системного вызова (потока A, если он сделал системный вызов, или потока B, если он продолжает работу);
   - если в результате обработки какой-либо процесс должен получить сигнал (сигнал отправлен обрабатываемым системным вызовом самому себе или другому процессу, или сигнал является результатом обработки ошибки или исключения, возникшей в потоке A), отмечает необходимость вызова зарегистрированного

или стандартного обработчика сигнала при следующем продолжении работы одного из потоков процесса, получившего этот сигнал;

- если потоком A было запрошено принудительное завершение другого процесса D (например, с помощью системного вызова kill — средствами диспетчера задач или иным способом), помечает его как завершаемый: когда дойдет время его исполнения, он будет завершен.

8. ОС завершает обработку ловушки
   - записывает в специальном защищенном регистре адрес, на который нужно передать управление — инструкции, на которой было ранее прервано исполнение потока A или B (если произошло переключение), или адрес обработчика сигналов, если именно он должен получить управление;
   - загружает значения регистров пользовательского пространства целевого процесса, в т. ч. таблицу страниц и указатель стека;
   - выполняет инструкцию возврата из обработчика ловушки.
9. Процессор завершает обработку ловушки — понижает уровень привилегий (выставляет целевой уровень) и передает управление по нужному адресу.
10. Теперь процессор выполняет ранее прерванный поток — A или B.

Данная схема описывает механизм в общих чертах. Заметим, что прерывания (аппаратные), ошибки и исключения могут возникнуть и при исполнении кода ядра, тогда ядро все равно переключится на их обработку. Ядро, однако, не может сделать системный вызов, т. к. тогда не произойдет корректной передачи аргументов и возврата значения, но может явно вызвать функции ядра, ответственные за выполняемые действия.

Глава 2. Основы архитектуры RISC-V

В данной главе описываются базовые представления об архитектуре RISC-V и ее краткий обзор: регистры, адресация, машинные инструкции, режимы привилегий, страничная адресация, системные вызовы. Описание дается в обзорно-ознакомительных целях и не претендует ни на полноту изложения архитектуры набора машинных команд, ни на полноценный учеб-

ник по программированию на Ассемблере RISC-V. С этой целью рекомендуется изучение спецификации архитектуры и специализированной литературы.

1. Andrew Waterman, Krste Asanović (Editors) «The RISC-V Instruction Set Manual». Volume I: User-Level ISA (набор инструкций непривилегированного режима), Volume II: Privileged Architecture (архитектура привилегированного режима). Распространяются в свободном доступе, в том числе в виде исходного кода по адресу `https://github.com/riscv/riscv-isa-manual` (дата доступа 01.02.2024).
2. David Patterson and Andrew Waterman «The RISC-V Reader: An Open Architecture Atlas».
3. SHAKTI Development Team «RISC-V ASSEMBLY LANGUAGE. Programmer Manual. Part I».

Изучение средств разработки на Ассемблере, возможно, например, по руководствам к соответствующим компиляторам. Документация по GNU Assembler размещена по адресу `https://sourceware.org/binutils/docs/as/` (дата доступа 01.02.2024), а также учебникам без привязки к архитектуре (в частности, учебникам по архитектуре x86-64, широко представленным на рынке, или туториалам, например `https://cs.lmu.edu/~ray/notes/gasexamples/`, дата доступа 01.02.2024).

§2.1. Об архитектуре RISC-V

CISC и RISC архитектуры

Исторически архитектуры набора машинных команд процессора проектировались с расчетом в том числе на прямое программирование системных и прикладных программ на Ассемблере, а также на облегчение создания компиляторов языков программирования высокого уровня — поддержку соответствующих абстракций. Это привело к включению в набор команд большого количества разнообразных инструкций. В последствии выяснилось, что такой подход сопряжен с рядом недостатков и обстоятельств, которые стимулировали частичный или полный отказ от него.

- Во-первых, многообразие инструкций усложняет аппаратную конструкцию процессора и делает его более дорогим и медленным, с более высоким энергопотреблением.
- Во-вторых, присутствие в наборе инструкций с различной длиной кода и различным временем исполнения (количеством циклов или так-

тов работы процессора, необходимых для их исполнения), делает невозможным прямую реализацию ***вычислительного конвейера*** — *параллелизма процессора на уровне машинных инструкций, при котором разные этапы исполнения инструкций исполняются на различных узлах.*

Например, если исполнение инструкции состоит из таких операций, как ***извлечение инструкции из памяти (IF)***, ***декодирование инструкции (ID)***, ***обращение к памяти (MEM)***, ***исполнение инструкции (EXEC)***, ***запись результата (WB)***, и каждая из них выполняется отдельным узлом, то первый узел IF может начинать извлекать вторую инструкцию потока выполнения сразу же после того, как будет извлечена первая и передана второму узлу (ID) — и так далее для других узлов. Таким образом, блоки обработки в параллельном режиме получают инструкцию от предыдущего блока (или из памяти), выполняют свою работу с ней и передают следующему блоку (или заканчивают работу) — по принципу работы производственного конвейера. Однако, конвейерная обработка эффективно реализуется, только если все инструкции имеют одинаковую длину, одинаковое количество этапов обработки, а на каждый этап затрачивается одинаковое количество тактов. Дополнительно на конвейере одного ядра с достаточно большим количеством узлов можно параллельно исполнять несколько потоков, эта технология называется ***суперскалярность*** (***гиперпоточность***).

- В-третьих, практика показала, что в реальных программах, в том числе в коде, создаваемом компиляторами языков программирования высокого уровня, значительная часть инструкций набора, особенно сложные и медленно работающие, не используются, отдается предпочтение более эффективным инструкциям.

Это стимулировало разработку архитектур, с сокращенным набором инструкций — RISC (Reduced Instruction Set Computer), в свою очередь классические архитектуры получили название CISC (Complex Instruction Set Computer) — составной (сложный, полный) набор инструкций. Однако полного отказа от CISC-архитектур не произошло в т. ч. в целях сохранения обратной совместимости. Так, архитектура x86_64, продолжающая совместимость с 16-ти и 32-х разрядными архитектурами процессоров 8086 фирмы Intel и более поздних, а также z/Architecture являются CISC. В то же время для получения преимущества от RISC-решений (конвейеризация, суперскалярность) современные процессоры используют RISC-ядро с закры-

тым набором низкоуровневых инструкций, на котором исполняется микрокод, интерпретирующий высокоуровневые CISC-инструкции. Более новые архитектуры, такие как ARM, Power и MIPS, исходно являются RISC-архитектурами.

Основными свойствами RISC-архитектуры являются:

- фиксированная длина кода машинных инструкций;
- большое количество регистров общего назначения;
- все операции изменения (арифметические, логические) выполняются только над содержимым регистров, используются отдельные операции над памятью для ***загрузки (чтения, load)*** и для ***сохранения (помещения, store)***: поэтому вместо RISC также такая архитектура может называться «***load-and-store***» в противоположность архитектурам, имеющим операции, выполняющие сразу три действия — «загрузить-изменить-записать;
- отсутствие или минимальное количество микрокода внутри процессора.

Заметим, что наличие такого микрокода является потенциальным источником ошибок и уязвимостей, которые при исполнении особых (***атакующих***) последовательностей байтов могут привести к обходу средств защиты процессора, таких как пользовательский режим и изоляция виртуальной памяти.

При этом *количество* инструкций не является характеристикой RISC-архитектуры (оно может быть даже больше, чем на CISC). Архитектуры с минимальным количеством инструкций носят название MISC (Minimal Instruction Set Computer).

Архитектура RISC-V

RISC-V (Reduced Instruction Set Computing — Five, на русском может произноситься как «риск-пять» или «риск-файв» и не должна путаться с отличной от нее архитектурой RISC-5. Буква V — римская цифра 5 — относится к пятой из предложенных при ее создании модификаций) — это открытая архитектура процессора, предоставляющая инфраструктуру для разработки процессоров.

Разработка RISC-V началась в 2010 году в Университете Беркли (в Калифорнии). RISC-V была разработана с учетом идеи открытости (свободы распространения, общедоступности). В 2015 году была создана RISC-V Foundation с целью стандартизации архитектуры и вывода на рынок стабильного продукта. С 2020 года ассоциация базируется в Швейцарии и именуется

RISC-V International. В настоящее время архитектура RISC-V привлекла внимание ряда компаний, учреждений и организаций, использующих данную архитектуру в различных разработках, от встраиваемых систем до серверов, а также в учебных и академических целях. В России развитием и популяризацией архитектуры занимается Альянс RISC-V (ссылка на сайт `https://riscv-alliance.ru/`, дата доступа 01.02.2024).

К основным достоинствам архитектуры RISC-V относят следующие.

- Отсутствие лицензионных ограничений позволяет любому желающему производителю реализовать процессор или другое устройство на данной архитектуре без внесения авторских или патентных ограничений.
- Относительно малый и простой набор инструкций, обеспечивающий сравнительную легкость как ее изучения, так создания эффективных процессоров и программных решений.
- RISC-V имеет базовый набор инструкций и наборы опциональных расширений. При реализации конкретного процессора производители могут выбирать только те расширения, которые им необходимы в конкретном устройстве. Также поддерживаются как 32-битные, так 64-битные модификации. Вариации RISC-V подходят для устройств практически всех типов, включая встроенные микроконтроллеры и суперкомпьютеры. Наличие базового неизменного набора инструкций обеспечивает стабильность архитектуры, а расширяемость и открытость обеспечивает минимизацию рисков того, что развитие и поддержка архитектуры будут прекращены.

Ряд этих и других фактов сделал архитектуру RISC-V перспективной для широкого распространения и использования. В настоящее время ее популярность растет.

Модификации RISC-V

Архитектура RISC-V представлена в двух модификациях: 32-битной и 64-битной. Базовый набор инструкций включает в себя операции с целыми числами соответствующей разрядности, называемые, соответственно, RV32I и RV64I. Данные инструкции включают в себя такие, как

- чтение и запись данных в оперативной памяти (инструкции загрузки и сохранения);
- битовый сдвиг, сложение и вычитание, логические операции, операции сравнения;

- операции перехода, системного вызова и инструкции отладки (необходимые для работы отладчиков ПО);
- привилегированные инструкции чтения/записи управляющих регистров.

Среди существующих расширений архитектуры (дополнительных возможностей) отметим следующие.

- Расширения для умножения и деления целых чисел (RV32M и RV64M). Обычно операции умножения и деления сложнее (с алгоритмической точки зрения, т. е. выполняются медленнее), чем операции сложения и вычитания, что затрудняет включение их в архитектуру типа RISC.
- Атомарные операции (RV32A и RV64A) — выполняющиеся неделимым образом, без риска вмешательства со стороны других ядер (процессоров). Необходимы для аппаратной реализации механизмов синхронизации.
- Расширения для операций над числами с плавающей точкой (RV32F и RV64F для чисел одинарной точности, и RV32D и RV64D для чисел двойной точности).
- Векторные расширения (RV32V и RV64V) — для операций над векторными данными, т. е. над массивами целиком).
- Криптографическое расширение для аппаратной поддержки алгоритмов шифрования.

Стандартная длина кода инструкций составляет 32 бита. Также существует расширение, включающее «сжатые» инструкции, занимающие 16 битов, поддерживающие возможности базового набора инструкций. Более короткий размер инструкций важен для экономии ресурсов (размера используемой под код программы оперативной памяти, кеша процессора и других), что может быть критично для маломощных встроенных систем.

Особенности адресации, представления и выравнивания данных

Машинные инструкции представляют собой битовые последовательности (двоичный машинный код), но для их человекочитаемости обычно используются текстовые (алфавитно-цифровые) мнемокоды машинных инструкций. Для перевода их в машинный код требуется трансляция. Не вдаваясь в детали двоичного машинного кода, в данном пособии приводятся только мнемокоды основных инструкций.

Все инструкции имеют операнды (аргументы), в качестве которых могут выступать имена регистров или непосредственные значения (***immediate***) — числа, включаемые в код инструкции. Трансляторы могут поддерживать задание мнемокода непосредственного значения в десятичном (например, `26`), шестнадцатеричном (с префиксом `0x`, например `0x1A`) или двоичном (с префиксом `0b`, например, `0b11010`) виде. В описании синтаксиса инструкций пояснения к возможным значениям операндов (вид регистра или непосредственное значение) будет показано курсивом.

Непосредственное значение может представлять собой как собственно число, так и адрес ячейки памяти, содержащей данные (для инструкций загрузки данных из памяти или сохранения данных в память) или код (для инструкций перехода). При программировании машинных инструкций адрес должен быть вычислен вручную. В отличие от программирования мнемокодов, Ассемблер (Assembler, транслятор языка ассемблера) вычисляет адреса данных в памяти автоматически по метке, которой снабжается инструкция или переменная. Метка инструкции может служить в т. ч. и именем функции в библиотеке (объектном модуле), код которой начинается с данной метки.

С точки зрения процессора машинная инструкция включает в себя и выполняемое действие (собственно инструкцию) и операнды. Из-за особенности концепции архитектуры RISC длина кода инструкции фиксируется, поэтому длина непосредственного значения ограничивается не размером регистра, а меньшей разрядностью. Конкретное ограничение зависит от инструкции, используемого расширения и варианта разрядности процессора, и может составлять 12 битов (для RV32I), 32 бита (для RV64I) или другую величину. Это ограничение относится как к непосредственному числовому значению, так и к непосредственному значению адреса. Для операции с более длинными значениями следует комбинировать несколько инструкций. Это может показаться неудобным для программирования. Поэтому помимо собственно машинных инструкций существуют более удобные в применении псевдоинструкции, раскрываемые в одну или несколько машинных инструкций.

Из-за особенностей аппаратной реализации при указании размера данных используется термин «***слово***» (***word***, ***машинное слово***). Количество байтов в слове зависит от архитектуры и разрядности процессора. Для RISC-V слово всегда составляет 4 байта (32 бита) — и для RV32I и для RV64I. Соответственно используются термины ***полуслово*** (***half word***) — 2 байта и ***двойное слово*** (***double word***) — 8 байтов. Таким образом, для операций над данными — например, ***загрузки*** (***чтения***, ***load***) данных из памяти в регистр и

сохранения (***помещения***, ***store***) данных из регистра в память — используются разные инструкции для данных разной длины (байт, полуслово, слово, двойное слово). При этом, так как регистры в RV32I составляют одно слово, инструкции для операций над двойным словом для 32-разрядной модификации не предусмотрены.

Кроме различных инструкций для данных различной длины могут использоваться различные инструкции для знаковых и беззнаковых значений, там где это существенно. RISC-V использует обратный дополнительный код для отрицательных значений, то есть все биты абсолютного значения отрицательного числа инвертируются и к результату добавляется 1. Например, значение `-5` длины в полуслово будет представлено как

```
11111111 11111011
```

а положительное число 5 представляется как

```
00000000 00000101
```

При этом значение, которое в знаковом случае трактуется как `-5` (полуслово) будет прочитано как `65531` в беззнаковом случае. При дополнении нулями слева (старшие биты) до слова будет получено значение

```
00000000 00000000 11111111 11111011
```

которое в обоих случаях трактуется как `65531`, а при дополнении единицами слева до слова — значение

```
11111111 11111111 11111111 11111011
```

которое в знаковом случае интерпретируется как исходное `-5`. Это показывает одну из причин различия инструкций для знаковых и беззнаковых значений — необходимо корректно дополнять данные до нужной длины нулями или единицами.

Также следует учитывать, что архитектура RISC-V использует обратный порядок байтов для хранения данных в памяти (***little-endian***) — от младшего байта к старшему. Так, если поместить в память из регистра слово, содержащее значение 4567

```
00000000 00000000 00010001 11010111
```

в памяти оно будет записано как

```
11010111 00010001 00000000 00000000
```

При этом терминология, относящаяся к битам и байтам регистра (старшие, младшие) предполагает, что в самих регистрах значения записаны в прямом порядке байтов. При сохранении значения, которое короче чем обрабатываю-

щая инструкция, его можно в дальнейшем корректно загрузить инструкцией для более короткого значения как в знаковом, так и в беззнаковом случае: например, если с помощью инструкции сохранения в память слова, обработать число, для записи которого хватает 16 битов (полуслово), его можно будет корректно загрузить в регистр с помощью инструкции загрузки полуслова.

Еще одним важным требованием к адресации данных и инструкций в памяти является ***выравнивание*** (***align***) ***по длине***. Выравнивание, например, по словам, означает, что адрес значения или инструкции должен быть кратен одному машинному слову (т. е. 4 байтам). На практике выравнивание обычно обеспечивается компиляторами языков высокого уровня или явно при программировании. Стандарт языка Си предполагает возможность автоматического выравнивания полей структуры путем вставки между ними дополнительных байтов (***padding***) так, чтобы адрес очередного поля начинался с нового слова, поэтому размер структуры может быть не равен сумме размеров полей. Автоматическое выравнивание элементов массива не предусмотрено, т. е. размер массива должен быть равен сумме размеров элементов.

Архитектура RISC-V требует, чтобы инструкции (адреса перехода) были выровнены по полусловам (т. е. были кратными двум). Явного требования для выравнивания значений в памяти не предусмотрено, однако выравнивание по словам обеспечит более высокую скорость доступа к данным. Дело в том, что обычно аппаратно за одно обращение к памяти осуществляется адресация и операция над словами целиком (выравненными), а не отдельными байтами. Доступ по невыравненному адресу, особенно к данным, пересекающим границы слов, может потребовать несколько обращений к памяти, поэтому отказ от выравнивания может замедлить программу в несколько раз.

§2.2. Инструкции пользовательского режима RISC-V

Регистры

Для выполнения непривилегированных инструкций архитектура RISC-V использует 33 основных регистра:

- регистр `x0`, значение которого всегда равно `0` и не может быть изменено (удобен для выполнения некоторых операций);
- 31 регистр с названиями `x1`, `x2`, …, `x32`, которые могут быть использованы в качестве регистров общего назначения (с точки зрения при-

меняемых операций), но некоторые из них имеют специальное применение согласно *конвенции о вызовах* (описанной ниже);

- регистр pc, выполняющий роль указателя инструкций.

Хотя регистры x1 — x31 могут применяться в любых целях и правила архитектуры не выделяют специальные регистры, например, в качестве указателя стека или сегмента данных, существует ***конвенция (соглашение) о вызовах***, регулирующая, какие регистры следует использовать в каких целях при вызове функций, описанном на стр. 85. Для облегчения понимания кода, регистры помимо имен x1 — x32 имеют более специфичные псевдонимы, отражающие эту конвенцию. Имена и назначения регистров представлены в таблице.

Регистр	Псевдоним	Назначение согласно конвенции о вызовах
x0	zero	Жестко кодированный ноль (доступен только для чтения, не может быть изменен)
x1	ra	Адрес точки возврата из функции (функция должна передать управление по этому адресу по завершении)
x2	sp	Указатель на вершину стека, может изменяться во время выполнения функции
x3	gp	Указатель глобальных данных (доступных всему процессу)
x4	tp	Указатель данных потока многопоточного процесса
x5-x7	t0-t2	Регистры для временных значений, функции могут изменять их произвольно, при вызове другой функции значение регистра может быть изменено ею
x8	fp/s0	Указатель стекового фрейма (локальных данных текущего вызова функции), функция может изменять его значение соответственно своим нуждам (например, сохранить значение регистра sp перед его изменением), но должна восстановить значение перед завершением работы
x9	s1	Сохраняемый регистр, функции могут изменять их значения произвольно, но обязаны восстановить их значения по завершении
x10-x11	a0-a1	Аргументы или возвращаемые значения функций
x12-x17	a2-a7	Аргументы функций
x18-x27	s2-s11	Сохраняемые регистры, функции могут изменять их значения произвольно, но обязаны восстановить их значения перед завершением работы
x28-x31	t3-t6	Регистры для временных значений, функции могут изменять

		их произвольно, при вызове другой функции значение регистров могут быть изменены ею
pc		Указатель инструкции, изменяется автоматически при выполнении инструкций на длину инструкции, а также инструкциями перехода или при срабатывании ловушки на требуемое значение

Данные регистры являются целочисленными регистрами соответствующей разрядности (32 бита для RV32I и 64 бита для RV64I).

Машинные инструкции

Инструкции загрузки непосредственных данных в регистр. Имеют 2 операнда:

- *rd* — регистр, в который загружаются данные;
- *imm* — непосредственное значение (данные).

Инструкция ***загрузки*** (***load***) копирует в регистр *rd* данные, непосредственно заданные в коде инструкции. Из-за ограничения длины машинного кода инструкции длина непосредственного значения ограничена — она меньше длины регистра. Таким образом, с помощью таких инструкций записывается только верхние (старшие) 20 битов регистра (RV32I). Младшие байты должны быть добавлены с помощью инструкций сложения. Инструкция перехода добавляет непосредственное значение к указателю инструкции как старшие 20 битов, т. е. обеспечивает переход на заданное смещение относительно текущей позиции вперед или назад (если прибавляемое значение отрицательное).

Инструкции с пояснениями приведены в таблице:

Название	Синтаксис	Действие
Load Upper Immediate	`lui` *rd*, *imm*	Загрузить значение *imm* длиной до 20 битов в старшие биты регистр *rd*. Недостающие биты регистра дополняются нулями.
Add Upper Immediate to PC	`auipc` *rd*, *imm*	Добавить значение *imm* к регистру-счетчику инструкции `pc` и записать результат в регистр *rd*. Может использоваться для вычисления адреса инструкции относительно текущей позиции (например, для перехода).

Например

```
lui t2, 0x110
```

Загружает значение `0x110` в старшие 20 битов регистра t2, младшие 12 битов дополняются нулями, т. е. в регистр запишется значение `0x110000` (RV32I).

```
auipc t1, 0x110
```

Складывает значение `0x110000` со значением в регистре `pc` и записывает результат в регистр `t1`. Если в `pc` было значение `0x1A`, то результат в `t1` будет `0x11001A`.

Инструкции загрузки данных из памяти и сохранения данных в память. Имеют 3 операнда:

- *rd* — регистр для записи информации из памяти или сохранения информации в память;
- *imm* — непосредственное значение адреса, откуда считать или куда записать информации (трактуется как младшие биты адреса);
- *rs* — регистр, относительно значения которого отсчитывается значение адреса.

Таким образом, инструкции ***загрузки (load)*** копируют в регистр *rd* данные, расположенные в памяти по адресу *rs+imm*, а инструкции ***сохранения (store)*** записывают данные из регистра *rd* в память по адресу *rs+imm*.

Инструкции с пояснениями приведены в таблице:

Инструкции загрузки		
Инструкция	**Синтаксис**	**Действие**
Load Byte	`lb` *rd*, *imm* (*rs*)	Загрузить знаковое однобайтовое значение (дополняет до размера регистра так, чтобы знак был сохранен)
Load Byte Unsigned	`lbu` *rd*, *imm* (*rs*)	Загрузить беззнаковое однобайтовое значение (дополняет до размера регистра нулями)
Load Half-word	`lh` *rd*, *imm* (*rs*)	Загрузить знаковое значение длиной в полуслово
Load Half-word Unsigned	`lhu` *rd*, *imm* (*rs*)	Загрузить беззнаковое значение длиной в полуслово
Load Word	`lw` *rd*, *imm* (*rs*)	Загрузить значение длины в одно слово, в RV64I значение трактуется как знаковое
Load Word Unsigned	`lw` *rd*, *imm* (*rs*)	Загрузить беззнаковое значение длины в одно слово
Load Double-word	`ld` *rd*, *imm* (*rs*)	Загрузить знаковое значение длины

Unsigned		в двойное слово — только RV64I
Инструкции сохранения		
Инструкция	**Синтаксис**	**Действие**
Store Byte	`sb` *rd*, *imm* (*rs*)	Сохранить однобайтовое значение (8 младших битов регистра)
Store Half-word	`sh` *rd*, *imm* (*rs*)	Сохранить значение длиной в полуслово (16 младших битов регистра)
Store Word	`sw` *rd*, *imm* (*rs*)	Сохранить значение длиной в машинное слово (32 бита, младших для RV64I)
Store Double-word	`sd` *rd*, *imm* (*rs*)	Сохранить значение длиной в двойное машинное слово — только RV64I

Например

```
lhu t3, 0x10(fp)
```

Сохраняет значение длиной 16 байтов, расположенное по адресу, вычисляемому как значение в регистре `fp` (начало стекового фрейма текущей функции) плюс шестнадцатеричное значение 10, в младшие 16 битов регистра `t3`, дополняя недостающие биты нулями или единицами, в зависимости от знака записываемого значения. Адрес может трактоваться как виртуальный (подвергаться трансляции) или как физический, в зависимости от того, включена или нет страничная адресация (стр. 103).

Арифметические инструкции. Имеют 3 операнда:

- *rd* — регистр для записи результата;
- *rs1* — регистр, содержащий левый операнд;
- *rs2* — регистр, содержащий правый операнд или *imm* — непосредственное значение.

Инструкция	**Синтаксис**	**Действие**
Addition	`add` *rd*, *rs1*, *rs2*	Прибавляет значение регистра *rs1* к значению регистра *rs2* и записывает результат в регистр *rd*. Выполняет операцию с регистром целиком (32-битовыми числами на RV32I и 64-битовыми на RV64I).
Subtraction	`sub` *rd*, *rs1*, *rs2*	Вычитает из значения регистра *rs1* значение регистра *rs2* и записывает результат в регистр *rd*. Выполняет операцию с регистром целиком (32-битовыми числами на RV32I и 64-битовыми на RV64I).
Add Word	`addw` *rd*, *rs1*, *rs2*	Прибавляет значение регистра *rs1* к

		значению регистра *rs2* и записывает результат в регистр *rd*. Выполняет операцию с 32-битовыми числами (только для RV64I).
Subtract Word	`subw` *rd, rs1, rs2*	Вычитает из значения регистра *rs1* значение регистра *rs2* и записывает результат в регистр *rd*. Выполняет операцию с 32-битовыми числами (только для RV64I).
Add Immediate	`add` *rd, rs1, imm*	Прибавляет значение *imm* к значению регистра *rs1* и записывает результат в регистр *rd*. Выполняет операцию с регистром целиком (32-битовыми числами на RV32I и 64-битовыми на RV64I).
Add Word Immediate	`addwi` *rd, rs1, rs2*	Прибавляет значение *imm* к значению регистра *rs1* и записывает результат в регистр *rd*. Выполняет операцию с 32-битовыми числами (только для RV64I).

Например

```
add t0, t2, 0x10
```

Прибавляет `0x10` к значению регистра `t2` и записывает результат в регистр `t0`. Значение регистра `t2` не изменяется.

Инструкции сравнения. Имеют 3 операнда:

- *rd* — регистр для записи результата;
- *rs1* — регистр, содержащий левый операнд;
- *rs2* — регистр, содержащий правый операнд или *imm* — непосредственное значение.

Инструкция	**Синтаксис**	**Действие**
Set Less Then	`slt` *rd, rs1, rs2*	Выясняет, верно ли, что значение регистра *rs1* строго меньше значения регистра *rs2*, записывает результат (0 — неверно, 1 — верно) в регистр *rd*. Значения трактуются как знаковое целое.
Set Less Then Immediate	`slti` *rd, rs1, imm*	Выясняет, верно ли, что значение регистра *rs1* строго меньше значения *imm*, записывает результат (0 — неверно, 1 — верно) в регистр *rd*. Значения трактуются как знаковое целое.
Set Less Then	`sltu` *rd, rs1, rs2*	Выясняет, верно ли, что значение ре-

Unsigned		гистра *rs1* строго меньше значения регистра *rs2*, записывает результат (0 — неверно, 1 — верно) в регистр *rd*. Значения трактуются как беззнаковое целое.
Set Less Then Immediate Unsigned	`sliu` *rd, rs1, imm*	Выясняет, верно ли, что значение регистра *rs1* строго меньше значения *imm*, записывает результат (0 — неверно, 1 — верно) в регистр *rd*. Значения трактуются как беззнаковое целое.

Например

```
slti t0, s1, 20
```

Если значение регистра `s1` строго меньше 20 (десятичное значение) записывает в регистр `t0` значение 1, в противном случае записывает значение 0.

Битовые инструкции. Имеют 3 операнда:

- *rd* — регистр для записи результата;
- *rs1* — регистр, содержащий левый операнд;
- *rs2* — регистр, содержащий правый операнд или *imm* — непосредственное значение.

Инструкции битового сдвига		
Инструкция	**Синтаксис**	**Действие**
Shift Left Logical Shift Left Logical Word	`sll` *rd, rs1, rs2* `sllw` *rd, rs1, rs2*	Выполняет битовый сдвиг влево значения регистра *rs1* на значение регистра *rs2*, дополняя справа битами 0, записывает результат в регистр *rd*. Word-версия выполняет операцию не над всем регистром, а над словом — 32-битами (только RV64I).
Shift Left Logical Immediate Shift Left Logical Immediate Word	`slli` *rd, rs1, imm* `slliw` *rd, rs1, imm*	Выполняет битовый сдвиг влево значения регистра *rs1* на значение *imm*, дополняя справа битами 0, записывает результат в регистр *rd*. Word-версия выполняет операцию не над всем регистром, а над словом — 32-битами (только RV64I).
Shift Right Logical Shift Right Logical Word	`srl` *rd, rs1, rs2* `srlw` *rd, rs1, rs2*	Выполняет битовый сдвиг влево значения регистра *rs1* на значение регистра *rs2*, дополняя слева битами 0 или 1 в зависимости от знака значения *rs1*, записывает результат в регистр *rd* (беззнаковый сдвиг). Word-версия выполняет операцию не над

		всем регистром, а над словом — 32-битами (только RV64I).
Shift Right Logical Immediate Shift Right Logical Immediate Word	`srli` *rd, rs1, imm* `srliw` *rd, rs1, imm*	Выполняет битовый сдвиг вправо значения регистра *rs1* на значение *imm*, дополняя слева битами 0 или 1 в зависимости от знака значения *rs1*, записывает результат в регистр *rd* (беззнаковый сдвиг). Word-версия выполняет операцию не над всем регистром, а над словом — 32-битами (только RV64I).
Shift Right Arithmetic Shift Right Arithmetic Word	`sra` *rd, rs1, rs2* `sraw` *rd, rs1, rs2*	Выполняет битовый сдвиг вправо значения регистра *rs1* на значение регистра *rs2*, дополняя слева битами 0 или 1 в зависимости от знака значения *rs1*, записывает результат в регистр *rd* (знаковый сдвиг). Word-версия выполняет операцию не над всем регистром, а над словом — 32-битами (только RV64I).
Shift Right Arithmetic Immediate Shift Right Arithmetic Immediate Word	`sra` *rd, rs1, rs2* `sraw` *rd, rs1, rs2*	Выполняет битовый сдвиг вправо значения регистра *rs1* на значение регистра *rs2*, дополняя слева битами 0 или 1 в зависимости от знака значения *rs1*, записывает результат в регистр *rd* (знаковый сдвиг). Word-версия выполняет операцию не над всем регистром, а над словом — 32-битами (только RV64I).
Инструкции битовой логики		
Инструкция	**Синтаксис**	**Действие**
AND AND Immediate	`and` *rd, rs1, rs2* `andi` *rd, rs1, imm*	Применяет побитово операцию логического *И* к значению регистра *rs1* и значению регистра *rs2* / значению *imm* записывает результат в регистр *rd*.
OR OR Immediate	`or` *rd, rs1, rs2* `ori` *rd, rs1, imm*	Применяет побитово операцию логического *ИЛИ* к значению регистра *rs1* и значению регистра *rs2* / значению *imm* записывает результат в регистр *rd*.
XOR XOR Immediate	`xor` *rd, rs1, rs2* `xori` *rd, rs1, imm*	Применяет побитово операцию логического разделительного ИЛИ к значению регистра *rs1* и значению регистра *rs2* / значению *imm* записывает результат в регистр *rd*.

Например

```
xori t0, s1, 0b00001010
```

Если значение регистра `s1` строго меньше 20 (десятичное значение) записывает в регистр `t0` значение 1, в противном случае записывает значение 0.

Инструкции ветвлений и переходов. Имеют до 3 операндов:

- *rs1* — регистр, содержащий левый операнд;
- *rs2* — регистр, содержащий правый операнд;
- *imm* — значение адреса, на которое происходит переход (относительно значения регистра pc).

Значение *imm* как обычно ограничено по длине (20 битов для RV32I). Оно расширяется с сохранением знака до размера регистра, умножается на 2 и добавляется к регистру. Таким образом достигается выравнивание адреса инструкции по полусловам.

Инструкции ветвления		
Инструкция	**Синтаксис**	**Действие**
Branch EQual Branch Not Equal	`beq` *rs1, rs2, imm* `bne` *rs1, rs2, imm*	Сравнивает значения регистров *rs1* и *rs2*, выполняет переход, если они равны (beq) или не равны (bne) между собой.
Branch Less Then Branch Greater or Equal	`blt` *rs1, rs2, imm* `bge` *rs1, rs2, imm*	Выполняет переход, если значение регистров *rs1* строго меньше (blt) или больше либо равно (bge) значению регистра *rs2*. Значения трактуются как знаковые целые.
Branch Less Then Unsigned Branch Greater or Equal Unsigned	`bltu` *rs1, rs2, imm* `bgeu` *rs1, rs2, imm*	Выполняет переход, если значение регистров *rs1* строго меньше (blt) или больше либо равно (bge) значению регистра *rs2*. Значения трактуются как беззнаковые целые.
Инструкции безусловного перехода и вызова		
Инструкция	**Синтаксис**	**Действие**
Jump and Link Jump and Link Register	`jal` *rd, imm* `jalr` *rd, rs1, imm*	Вызывает функцию по указанному адресу. Адрес возврата — значение регистра pc — сохраняется в регистре *rd*. Инструкция jal вычисляет адрес функции как удвоенное значение *imm*, jalr — как значение регистра *rs1* плюс *imm*.
Environment CALL	`ecall`	Выполняет системный вызов.
Environmnent BREAK	`ebreak`	Вызывает события точки прерывания отладчика (***breakpoint***) — отладчик должен вставить данные инструкции в код отлаживаемого приложения.

Например

```
addi t1, zero, 0x10

loop: # метка инструкции для перехода:
      # код тела цикла помещается здесь
      subi t1, t1, 1 # итератор цикла
bne t1, zero, loop   # условный переход
```

Выполняет цикл: регистр t1 пробегает значения от 16 до 0. Изначальное значение выставлено как сумма регистра zero (x0), доступного только значения и всегда имеющего значение 0, а также явного значения 16. Начало цикла помечено меткой `loop` для автоматического вычисления адреса точки перехода. В конце выполнения тела цикла происходит вычитание из значение регистра t1 явного значения 1 с записью в тот же регистр. Последняя инструкция осуществляет переход по адресу `loop`, если значение t1 не равно значению регистра zero (0).

Псевдоинструкции

Для облегчения программирования могут использоваться псевдоинструкции, раскрывающиеся компилятором в последовательности одной или нескольких машинных инструкций. Приведем некоторые из них в виде таблицы.

Пседвоинструкции загрузки и сохранения		
Инструкция и синтаксис	**Раскрытие**	**Действие**
MoVe `mv` *rd*, *rs*	`addi` *rd*, *rs*, `0`	Помещает значение регистра *rs* в регистр *rd*.
Load Immediate `li` *rd*, *imm*	`lui` *rd*, *imm[31:12]* `addi` *rd*, *rd*, *imm[11:0]*	Загружает явное значение *imm* в регистр *rd*, разбивая его на старшие 20 битов и младшие 12 битов (RV32I), тем самым облегчая соблюдение ограничения на длину непосредственного значения. Аналогично раскрывается в RV64I
Load Address `la` *rd*, *imm*	`lui` *rd*, *imm[31:12]* `addi` *rd*, *rd*, *imm[11:0]*	Загружает значение адреса *imm* в регистр *rd*, разбивая его на старшие 20 битов и младшие 12 битов (RV32I), тем самым облегчая соблюдение ограничения на дли-

		ну непосредственного значения. Аналогично раскрывается в RV64I
Арифметические и логические псевдоинструкции		
Инструкция и синтаксис	**Раскрытие**	**Действие**
NEGate `neg` *rd*, *rs*	`sub` *rd*, `x0`, *rs*	Помещает в регистр *rd* значение, противоположное записанному в *rs*.
NOT `not` *rd*, *rs*	`xori` *rd*, *rs*, -1	Побитовое отрицание значения регистра *rs*, результат запивает в *rd*.

Set if EQual to Zero
`seqz` *rd*, *rs*
 	
Set if Not EQual to Zero
`snez` *rd*, *rs*
	`sltiu` *rd*, *rs*, `1`
	
	`sltu` *rd*, *rs*, `1`
		Помещает в регистр *rd*, значение 0 или 1, в зависимости от того равно 0 или нет значение регистра *rs*.

Set if Less Then Zero `sltz` *rd*, *rs* Set if Greater Than Zero `sgtz` *rd*, *rs*	`slt` *rd*, *rs*, `x0` `sltu` *rd*, `x0`, *rs*	Помещает в регистр *rd*, значение 0 или 1, в зависимости от меньше или больше 0 (строго) значение регистра *rs*.
Псевдоинструкции ветвления		
Инструкция и синтаксис	**Раскрытие**	**Действие**
Branch if Equal to Zero `beqz` *rs*, *imm* Branch if Equal to Zero `beqz` *rs*, *imm*	`beq` *rs*, `x0`, *imm* `bne` *rs*, `x0`, *imm*	Осуществляет переход, если значение регистра *rs* равно (beqz) или не равно (bnez) 0.
Branch if Less or Eq. to 0 `blez` *rs*, *imm* Branch if Greater or Eq. to 0 `bgez` *rs*, *imm* Branch if Less Then Zero `bltz` *rs*, *imm* Branch if Greater Then Zero `bgtz` *rs*, *imm*	`bge x0`, *rs*, *imm* `bge` *rs*, `x0`, *imm* `blt` *rs*, `x0`, *imm* `blt x0`, *rs*, *imm*	Осуществляет переход, если значение регистра *rs*, соответственно меньше либо (blez) равно, больше либо (bgez) равно, строго меньше (bltz) или строго больше (bgtz) 0.
Branch if Greater Then `bgt` *rs1*, *rs2*, *imm* Branch if Less of Equal `ble` *rs1*, *rs2*, *imm*	`blt` *rs2*, *rs1*, *imm* `bge` *rs2*, *rs1*, *imm*	Осуществляет переход, если значение регистра *rs1* строго больше (bgt) или меньше либо равно (ble) значению регистра *rs2*. Дополняют симметричные им инструкции bge и blt.

Псевдоинструкции перехода		
Инструкция и синтаксис	**Раскрытие**	**Действие**
RETURN `ret`	`jalr x0, x1, 0`	Возврат из функции — переход по адресу, сохраненному в регистре x1 (ra) согласно конвенции. При этом адрес возврата никуда не сохраняется, т. к. регистр x0 (zero) доступен только для чтения.
Jump `j` *imm* Jump Register `jr` *rs*	`jal x0,` *imm* `jalr x0,` *rs*`, 0`	Выполняет переход по явно заданному адресу *imm* (j) или по адресу, заданному в регистре *rs* (jr).
Другие псевдоинструкции		
Инструкция и синтаксис	**Раскрытие**	**Действие**
No Operaton `nop`	`addi x0, x0, 0`	Не выполняет никаких действий. Может использоваться для замещения инструкции, там где она предполагается, но выполнения действия не предусмотрено.

Сегменты программы и их использование

Хотя теоретически вся программа, т. е. ее код, данные и стек, может быть представлена в виде одного фрагмента кода, операционные системы и процессорные архитектуры обычно раздельно управляют соответствующими ***сегментами*** (***segment***), выделяя под них независимые (непересекающиеся) участки в памяти. Таким образом, каждой программе выделяется

- ***текстовый сегмент*** (***text segment*** или ***сегмент кода***, ***code segment***) — *содержит собственно исполняемый код программы* (двоичный): термин «текстовый» происходит от того, что содержимое данного сегмента формируется из исходного текста программы;
- ***сегмент данных (data segment)*** *используется для хранения глобальных данных и статических переменных функций программы;*
- ***сегмент стека (stack segment)*** *используется для хранения локальных данных и параметров функций программы, а также данных, необходимых для управления вызовами функции.*

Так как каждый поток исполняет свою последовательность инструкций и может вызывать различные функции, у многопоточного процесса для каж-

дого потока должен быть выделен свой сегмент стека, в то время как текстовый сегмент и сегмент данных могут быть общими.

За выделение данных сегментов отвечает операционная система, поэтому не существует специальных машинных инструкций для «назначения» данных сегментов. Соответствующая информация размещается в исполняемом файле программы или библиотеки для разделения сегментов программы, кода функций и глобальных переменных. Там же указываются имена функций, переменных и другая информация, необходимая для динамической компановки, загрузки в память и запуска процесса программы. Формат такого файла специфичен для операционной системы и является частью ее ***двоичного интерфейса приложений (ABI)***. В операционных системах на базе Linux используется формат ***ELF*** *(формат исполняемых и компонуемых файлов)*. Традиционно исполняемые файлы этого формата не имеют расширения, а файлы библиотек используют расширение `.so`.

При использовании языка ассемблера могут использоваться специальные ***директивы*** *— средства языка, предназначенные для определения части сегмента, которые при компиляции и компоновке будут преобразованы в соответствующие коды формата исполняемого файла. Эти директивы специфичны для компилятора* Ассемблера, но обычно должны присутствовать две основные директивы — .text и .data (или с аналогичными названиями) — ***директивы обозначения начала текстового сегмента и сегмента данных соответственно***. Сегмент стека обычно резервируется операционной системой автоматически.

Не существует специальных машинных инструкций для объявления и определения переменных. Определение переменной в памяти состоит в резервировании соответствующего количества байтов в сегменте данных или в сегменте стека. Имя переменной (так же как и функции) с точки зрения машинного кода представляет собой ее *адрес (виртуальный)*, который может задаваться как *абсолютное значение* или как *относительное значение*, например, относительно начала сегмента данных или относительно вершины стека. Программирование в машинных кодах требует ручного вычисления данного адреса. Язык ассемблера использует, во-первых, ***метки*** для инструкций и директив, автоматически заменяемые на адрес (относительный или абсолютный) или ссылку на компонуемый объект (в другом объектном модуле или библиотеке) соответствующей ячейки памяти, во-вторых — ***директивы определения данных***, обеспечивающих резервирование необходимого количества байтов в памяти и, при необходимости, их инициализацию

начальными значениями. К таким директивам, например, могут относиться следующие:

Директива	Описание	Пример
.byte	Определяет байт данных	`.byte 13` (1 байт с десятичным значением 13)
.half	Определяет половину слова (2 байта)	`.half 31267` (2 байта с десятичным значением 31267)
.word	Определяет слово (4 байта)	`.word 0xB14A1` (4 байта с шестнадцатеричным значением B14A1)
.dword	Определяет двойное слово (8 байтов)	`.dword` (8 байтов без инициализации)
.asciz	Определяет нуль-терминированную (ASCII-Z) строку	`.asciz "Hello, world"` (13 байтов с кодами соответствующих символов и кодом 0 в конце)

Таким образом, для определения глобальных данных, явных значений и статических переменных следует использовать подобный ассемблерный код:

```
.data                       # Начало сегмента данных
    val:  .word 231         # Целочисленная переменная
    msg:  .asciz "Hello"    # Строковая переменная msg
    r:    .dword 5          # Переменная-двойное слово
```

После этого в текстовом сегменте можно использовать указанные метки в качестве значений, например

```
.text           # Начало текстового сегмента
  # Необходимый код здесь
  lw a0, val    # Загружает значение переменной val
                # длиной в 1 слово в регистр a0
  la a1, msg    # Загружает в регистр a1 адрес строки msg
  ld a2, r      # Загружает в регистр a2 значение
                # двойного слова в переменной val
```

При этом данный ассемблерный код, скорее всего, будет развернут в следующий код машинных псевдоинструкций:

```
lw a0, 0(gp)
la a1, 4(gp)
ld a2, 10(gp)
```

Такой код предполагается потому, что, во-первых, регистр gp используется в качестве указателя начала сегмента данных, и будет инициализирован системой соответственно. Во-вторых, относительные адреса переменных внутри

сегмента вычисляются компилятором Ассемблера автоматически в соответствии с их расположением: 0 для val (начало), 4 для msg (0 + длина val, т. е. 4 байта), 10 для r (4 плюс длина msg, т. е. 6 байтов). *Обратите внимание на разные инструкции для загрузки в регистр значений (целочисленных в данном примере) переменных и их адресов (в данном случае адреса строки — массива символов).*

Для сохранения локальных данных в сегменте стека не предусмотрено директив, автоматически вычисляющих их адрес. Вместо этого следует вручную вычислять адреса смещения относительно вершины стека. Операционная система инициализирует начальное значение вершины стека в регистр sp, которое в дальнейшем можно изменять средствами программы. Например, чтобы зарезервировать те же 18 байтов в стеке под 3 переменные, что и в предыдущем примере, необходимо выполнить

```
addi sp, sp, -18 # Резервирование 18 байтов в стеке
lw a0, 0(sp)   # Доступ к переменной в вершине стека
               # (ее длина 4 байта)
la a1, 4(sp)   # Доступ к адресу переменной со смещением
               # в 4 байтах от вершины
               # (ее длина 6 байтов)
ld a2, 10(sp)  # Доступ к переменной со смещением
               # в 10 байтах от вершины
               # (ее длина 8 байтов)
```

Например, подобный код может использоваться для определения локальных переменных функции или передачи параметров в функцию через стек. По окончании работы с данными переменными следует вновь изменить вершину стека:

```
addi sp, sp, 18  # Возврат исходного значения sp
```

Уменьшение вершины стека соответствует операции помещения значения (Push) в стек, увеличение — изъятию (Pop) из стека (без собственно копирования значения). Если один и тот же код будет исполнен при разном значении регистра sp, он обратится к разным виртуальным адресам в памяти, что обеспечивает корректную работу стека вызовов функций. Изменение значения регистра sp само по себе никак не влияет на содержание оперативной памяти (сегмента стека), не инициализирует и не зануляет данные, поэтому обращения возможны как с положительным, так и отрицательным смещением относительно его значения. Но корректное вычисление адреса переменных в соответствие с их длиной, так же как и учет того факта, что вызываемые

функции могут смещать границу стека вниз для выделения места под их стековый фрейм, и перезаписывать находящиеся там данные, критично.

Так как значение регистра sp может изменяться в ходе выполнения кода функции, т. е. переменные могут резервироваться в стеке во время работы, адреса смещений ранее зарезервированных переменных могут стать недействительными. Для облегчения работы можно сохранять начало фрейма функции в регистре fp.

При многопоточном программировании разные потоки одного процесса могут вызывать разные машинные инструкции и разные функции, поэтому каждый поток должен иметь свой, корректно инициализированный сегмент стека. Также каждый поток может резервировать свой локальный сегмент данных потока, адрес которого следует записывать в регистр tp.

Вызов функций, системные вызовы и соглашение о вызовах

Согласно конвенции (соглашению) о вызовах, для выполнения вызова функций следует поместить аргументы в регистры a0 — a7 и выполнить инструкцию JAL. Функция в свою очередь должна поместить выходные данные в регистры a0 — a1 и выполнить псевдоинструкцию ret. Например, код функции может быть таким

```
op_plus:            # метка кода в памяти –
                    # функция сложения a0 и a1
    add a0, a0, a1 # собственно сложение
    ret             # возврат
```

А вызов функции будет осуществляться следующим образом

```
li a0, 3          # Запись первого аргумента
li a1, 5          # Запись второго аргумента
jal ra, op_plus   # Вызов функции (по метке)
sw a0, n          # Запись результата в память (по метке)
```

Этому может соответствовать, например, такой Си код функции

```
int op_plus (int n, int m)
{
    return n + m;
}
```

и код вызова функции

```
n = op_plus (3, 5);
```

Реальный компилятор вряд ли создаст такой ассемблерный код из указанного Си-кода, так как будет применять различные технологии оптимизации. Приведенный пример показывает ассемблерный код вызова функции, осуществляемого в соответствии с соглашением о вызовах.

Смысл аргументов и возвращаемого значения, а также регистры для их передачи, зависят от конкретной функции. Инструкция jal перед осуществлением перехода автоматически сохраняет текущее значение регистра pc (указатель инструкции) в регистре ra, а псевдоинструкция ret раскраивается в инструкцию

```
jalr zero, ra, 0
```

что в точности обеспечивает возврат из функции к инструкции, следующей за инструкцией вызова функции.

Таким образом, в RISC-V точка возврата всегда передается через регистр, а не стек, что имеет ряд преимуществ, включая ускорение времени работы (не требуется обращение к памяти), а также обеспечивает некоторую защиту от риска переполнения стекового буфера и перезаписи значения адреса возврата, например, пользовательскими данными в результате программной ошибки. Такая ошибка может стать источником серьезной уязвимости в безопасности приложения, если враждебно настроенный пользователь подберет выходные данные таким образом, чтобы передача управления по выходу из функции была осуществлена непосредственно на эти данные, получив тем самым полный контроль над программой. Следует, однако, отметить, что, если функция сама вызывает другую функцию, то она должна предварительно сохранить значение регистра ra в стеке, и эта другая функция может его перезаписать из-за аналогичной ошибки. Поэтому такая защита не абсолютна, такая особенность архитектуры RISC-V защищает только частично и только от одной специфической уязвимости и не избавляет от необходимости контролировать размер буфера.

Благодаря наличию большого количества регистров, архитектура RISC-V позволяет передать через них до 8 целочисленных или адресных (указателей) аргументов функции. Для этого используются регистры a0 — a7. Временные локальные переменные функций также можно сохранять в регистры — для этого используются 7 регистров t0 — t6 и 13 регистров s0 — s12.

Использование регистров для хранения данных вместо оперативной памяти повышает производительность. Имеется ряд ситуаций, когда регистров недостаточно, либо их применение невозможно, тогда данные можно передавать через стек или по адресу (как указатель). Например, функция может использовать более 8 входных и выходных параметров (хотя, такие функции на

практике не рекомендуются к созданию). Для хранения больших данных (структуры, массивы, в т. ч. строки) следует использовать стек или сегмент данных памяти, но через регистры можно передать их адрес. Также регистры не являются общими между потоками выполнения процесса (так же как и стек вызовов), поэтому общие переменные потоков должны сохраняться в соответствующей области памяти. Кроме того, если функция меняет значения регистров s0 — s12, то по соглашению о вызовах она обязана восстановить их значения по завершении, т. е. необходимо сохранять их в стеке до изменения и восстановить значения по завершении. При вызове функции, наоборот, могут измениться значения регистров t0 — t6, поэтому перед вызовом функции следует позаботиться об их сохранении в стеке или использовании других регистров. Иными словами, по соглашению о вызовах

- ***вызывающий (caller)*** функцию должен как минимум позаботиться о *сохранении перед вызовом* и *восстановлении после вызова* значений регистров ra (адрес возврата), t0 — t6 (временные регистры) и a0 — a7 (аргументы), если их значения будут использоваться в дальнейшем (то есть эти регистры относятся к ***сохраняемым вызывающим***, ***caller-saved***);
- ***вызываемая (callee)*** функция (т. е. любая функция) должна обязательно позаботиться о *сохранении в начале исполнения* и *восстановлении перед возвратом* регистров sp (указатель стека), s0/fp (начало стекового фрейма) и s1 — s12 (сохраняемые регистры), если их значения изменялись внутри функции (то есть эти регистры относятся к ***сохраняемым вызываемым, callee-saved***).

Аналогичные правила соглашения существует и для регистров расширения архитектуры (например, регистров для работы с числами с плавающей точкой).

Значения регистров обычно сохраняются в стеке. Например, пусть вызывающему требуется сохранение регистров t0 — t3. Тогда это можно сделать следующим образом (на RV64I)

```
addi sp, sp, -24 # Смещение указателя стека (8 * 3 байта)
sd t0,  0(sp)    # Сохранение t0
sd t1,  8(sp)    # Сохранение t1
sd t2, 16(sp)    # Сохранение t2
# Здесь располагается код вызова функции
ld t0,  0(sp)    # Восстановление t0
ld t1,  8(sp)    # Восстановление t1
ld t2, 16(sp)    # Восстановление t2
```

```
addi sp, sp, 24  # Восстановление указателя стека
```

Теоретически компилятор (или программист на Ассемблере) может не придерживаться соглашения о вызовах и использовать регистры по своему усмотрению. Действительно, стандарт языка Си ничего не говорит о том, какие регистры следует использовать для передачи аргументов, какие и когда сохранять и восстанавливать, как использовать стек и т. д. Однако в этом случае получаемый двоичный код функций будет несовместимым. Компоновка программ, библиотек и объектных модулей, полученных несовместимыми компиляторами, приведет к непредсказуемому результату, даже если они будет записаны в одинаковом формате исполняемых файлов (например, ELF). Таким образом, *соглашение о вызовах, как и архитектура машинных команд, порядок байтов, используемый формат представления отрицательных чисел и другие особенности двоичного представления данных и кода, является частью двоичного интерфейса приложений (*ABI*)*.

Соглашение о вызовах относится не только к вызовам функций, но и к системным вызовам. Для осуществления системного вызова его аргументы записываются в регистры a0 — a6, номер системного вызова — в регистр a7, а возвращаемое значение записывается в регистр a0. Например, данный системный вызов write на ОС с ядром Linux

```
ret = syscall (__NR_write, 1, buffer, 1024);
```

может быть выполнен следующим кодом (реальный код в Linux будет включать в себя вызов ряда оберточных функций):

```
li a7, 64     # Запись номера системного вызова write
li a0, 1      # Запись номера файлового дескриптора
la a1, buffer # Запись адреса буфера вывода
li a2, 1024   # Запись объема выводимых данных
ecall         # Системный вызов
sd a0, ret    # Запись возвращаемого значения
              # в переменную ret
```

Возвращаемое значение системного вызова write — количество фактически записанных байтов или отрицательный код ошибки — будет записано в регистре a0, значение которого можно проверить после системного вызова. Параметры системных вызовов в реальных операционных системах практически всегда передаются посредством регистров, т. к. стеки для пространства ядра и пользовательского пространства разделяются во избежание влияния ошибок в коде процесса на работу ядра и других соображений безопасности (чтобы не создать риски доступности данных пространства ядра в пользовательском пространстве).

Атомарные операции

Состояние гонки *возникает при попытке одному или нескольким потокам (процессам, процессорам, ядрам процессора) обратиться к общему ресурсу, например, к общей переменной.* Если некоторое манипулирование общим ресурсом не является ***атомарным***, *то есть* ***неделимым*** *на промежуточные шаги*, то исполнение такой операции может быть прервано в результате переключения контекста, что может привести к состоянию гонки. Заметим, что даже операция инкремента не является атомарной. Код

```
++i;
```

может развернуться в следующие машинные инструкции

```
lw    a0, i     # Прочитать значение переменной в регистр
addw  a0, a0, 1 # Увеличить значение регистра на 1
sw    a0, i     # Записать значение регистра в переменную
```

Таким образом, если два потока будут пытаться примерно в одно и то же время выполнить операцию инкремента одной и той же переменной (на одном процессоре или на разных процессорах), то фактический порядок исполнения инструкций может оказаться произвольным, например таким

```
lw    a0, i      # (1 поток, i=1, a0=1)
addw  a0, a0, 1  # (1 поток, i=1, a0=2)
lw    a0, i      # (2 поток, i=1, a0=1)
addw  a0, a0, 1  # (2 поток, i=1, a0=2)
sw    a0, i      # (2 поток, i=2, a0=2)
sw    a0, i      # (1 поток, i=2, a0=2)
```

В результате вместо увеличения значения переменной i на 2 получится увеличение только на 1. *Для доступа к общему ресурсу требуется создание блокировки, «атомизирующей» операцию с ним.*

Наивная попытка организовать блокировку с помощью обычной переменной-замка не даст нужно эффекта.

```
int lock = 0  // Изначально замок открыт
// …
while (lock); // Ждем пока замок закрыт
lock = 1;     // Закрываем его
++i;          // Работаем с общим ресурсом
lock = 0;     // Открываем замок
```

Действительно, мало шансов, что на какой-либо архитектуре комбинация

```
while (lock); lock = 1;
```

будет раскрыта в одну машинную инструкцию, поэтому после того, как будет осуществлен выход из цикла, возможно закрытие замка другим потоком. Более того, *сами машинные инструкции по умолчанию не должны рассматриваться как атомарные* (особенно на CISC-архитектурах, требующих исполнения микрокода, но и на RISC-архитектурах это не гарантируется для всех инструкций). Вмешательство может произойти, например, в результате аппаратных прерываний, работы параллельных процессоров (ядер), необходимости разбить данные по размеру машинного слова для передачи по шине памяти, в том числе при доступе к невыравненным данным, и по другим причинам.

Для решения этой проблемы в современные архитектуры вводятся ***атомарные машинные инструкции***, для которых гарантируется, что их исполнение не будет прервано до полного завершения, а также то, что на время их исполнения до полного завершения производится блокировка шины памяти от вмешательства других процессоров. В RISC-V атомарные инструкции представлены в атомарном расширении архитектуры RV32A (32-битное) и RV64A (64-битное). Примерами таких инструкций являются следующие.

Инструкция	**Синтаксис**	**Действие**
Load Reserved Word Load Reserved Double Word	`lr.w` *rd, rs1* `lr.d` *rd, rs1*	Загрузить значение в одно слово / в двойное слово (RV64A) в регистр
SWAP Word SWAP Double word	`amoswap.w` *rd, rs1,* (*rs2*) `amoswap.d` *rd, rs1,* (*rs2*)	Атомарным образом загружает в регистр *rd* значение, находящееся по адресу, указанному в регистре *rs2,* меняет местами значения регистров *rd* и *rs1*, и сохраняет значение регистра rd по адресу, указанному в регистре *rs2:* по адресу в *rs2* записывается значение регистра *rs1*, в *rd* записывается старое значение, находившееся по адресу *rs2* (работает со знакомыми значениями соответствующей длины, регистры *rd* и *rs1* могут

		совпадать).
ADD Word Add Double word	`amoadd.w` *rd, rs1,* (*rs2*) `amoadd.d` *rd, rs1,* (*rs2*)	Записывает в *rd* сумму значений в регистре *rs1* и расположенном в памяти по адресу, записанному в регистре *rs2*.
XOR Word/Double word AND Word/Double word OR Word/Double word	`amoxor.w` *rd, rs1,* (*rs2*) `amoxor.d` *rd, rs1,* (*rs2*) `amoand.w` *rd, rs1,* (*rs2*) `amoand.d` *rd, rs1,* (*rs2*) `amoor.w` *rd, rs1,* (*rs2*) `amoor.d` *rd, rs1,* (*rs2*)	Выполняют соответствующие логические операции с операндом, записанном в регистре *rs1*, в памяти по адресу в регистре *rs2* и записью результата в регистр *rd*.
MINimim Word/ Double word MAXimum Word/ Double word	`amomin.w` *rd, rs1,* (*rs2*) `amomin.d` *rd, rs1,* (*rs2*) `amomax.w` *rd, rs1,* (*rs2*) `amomax.d` *rd, rs1,* (*rs2*)	Записывает в регистр *rd* минимум / максимум значений регистра *rs1* и значения в памяти, расположенном по адресу, заданному в регистре *rs2* (значения знаковые)
MINimim Unsigned Word/ Double word MAXimum Unsigned Word/Double word	`amominu.w` *rd, rs1,* (*rs2*) `amominu.d` *rd, rs1,* (*rs2*) `amomaxu.w` *rd, rs1,* (*rs2*) `amomaxu.d` *rd, rs1,* (*rs2*)	Аналогично для беззнаковых значений

Таким образом, для организации функций захвата и освобождения спинблокировки можно использовать инструкцию amoswap.w:

```
# Функция захвата блокировки,
# Параметр (регистр a0) - адрес переменной-замка
acquire_spinlock:
    # Загрузить значение 1 (занят) в регистр t0
    li, t0, 1
# Цикл попыток захвата
retry_acquire:
    # Записывает по адресу a0 значение в регистре t0 (1)
    # записывает предыдущее значение в t1 (атомарно!)
    amoswap.w t1, t0, (a0)
```

```
    # Если значение не было равно 0, т.е.
    # было заблокировано, - повторить попытку
    bnez t1, retry_acquire

    # Выход из функции
    ret


# Функция освобождения блокировки
# Параметр (регистр a0) - адрес переменной-замка
release_spinlock:
    # Записывает по адресу в регистре a0 значение 0
    # без сохранения старого значения
    # (запись в регистр zero игнорируется)
    amoswap.w zero, zero, (a0)

    # Выход из функции
    ret
```

Представленная реализация является очень схематичной, т. к., во-первых, может потребоваться некоторая, зависящая от конкретной платформы и ситуации корректировка используемых инструкций синхронизации (amoswap.w.aq и fence, которые регулируют завершение исполнения предшествующих операции загрузки-записи в конвейере, описание которых лежит за рамками пособия). Также разумным будет минимизировать использование потенциально медленной атомарной операции в первых попытках захватить блокировку:

```
# Функция захвата блокировки,
# Параметр (регистр a0) - адрес переменной-замка
acquire_spinlock:
    # Загрузить значение 1 (занят) в регистр t0
    li, t0, 1
retry_acquire:
    # Простая попытка захвата
    # Загрузить значение по адресу в регистре a0
    # (a0 плюс смещение 0) в регистр t1
    lw t1, 0(a0)
    # Если значение не было равен 0 (было захвачено),
    # повторить попытку
    bnez t1, retry_acquire
    # Здесь возможно (но не точно из-за гонки),
    # что значение было равен 0
    # Далее необходимая атомарная попытка захвата
```

```
    # Записывает по адресу a0 значение в регистре t0 (1)
    # записывает предыдущее значение в t1 (атомарно!)
    amoswap.w t1, t0, (a0)
    # Если значение не было равен 0, т.е.
    # было заблокировано, - повторить попытку
    bnez t1, retry_acquire
    # Выход из функции
    ret
```

Схема использования функций следующая:

```
.data
lock: .word 0     # Блокировка (переменная-замок)
                  # Инициализирован как разблокированная

.text
# Здесь обычный код
la a0, lock               # Запись адреса замка
jal ra, acquire_spinlock  # Вызов функции захвата
# Здесь код, требующий монопольного доступа к ресурсу
la a0, lock               # Запись адреса замка
jal ra, release_spinlock  # Вызов функции освобождения

# Здесь обычный код
```

Обращаем внимание на то, что использование спин-блокировок связано с риском потенциально длительного удержания процессора в цикле попыток захвата, поэтому для длительных операций следует прибегать к средствам операционной системы по приостановке выполнения потоков в ожидании освобождения блокировки (семафоров и других). Использование спин-блокировок требуется для реализации таких механизмов операционной системы, а также атомарных переменных и других средств синхронизации. Также спин-блокировки являются единственно доступными механизмами синхронизации там, где приостановка невозможна, например, в обработчиках прерываний.

Другие расширения

Другие расширения архитектуры RISC-V включают в себя, например, операции умножения и деления, операции для чисел с плавающей точкой, криптографические операции, векторные операции и другие. Рассмотрим только расширения умножения и деления RV32M и RV64M. Операции умножения и деления обычно сложнее алгоритмически и выполняются медленнее,

чем операции сложения и вычитания, поэтому они обычно не включаются в базовую архитектуру типа RISC.

К инструкциям умножения относятся такие инструкции, как

- `mul` *rd, rs1, rs2* — умножение *rs1* на *rs2* с записью результата в *rd*;
- `div` *rd, rs1, rs2* — деление *rs1* на *rs2* с записью результата в *rd*;
- `rem` *rd, rs1, rs2* — вычисление остатка от деления *rs1* на *rs2* с записью результата в *rd*;

работающие со знаковыми исходными значениями и использованием всего исходного регистра в качестве значения, а также инструкции, работающие с частью регистра

- `mulh` *rd, rs1, rs2* — умножение старших полуслов *rs1* на *rs2* с записью результата в *rd*;
- `mulw` *rd, rs1, rs2* — умножение старших слов *rs1* на *rs2* с записью результата в *rd* (только RV64M);
- `divw` *rd, rs1, rs2* и `remw` *rd, rs1, rs2* — деление и вычисление остатка от деления старших слов *rs1* на *rs2* с записью результата в *rd* (только RV64M);

инструкции, работающие с беззнаковыми значениями

- `divu` *rd, rs1, rs2* — беззнаковое деление *rs1* на *rs2* с записью результата в *rd*;
- `remu` *rd, rs1, rs2* — беззнаковый остаток от деления *rs1* на *rs2* с записью результата в *rd*;

и другие.

§2.3. Привилегированные инструкции RISC-V

Режимы работы и уровни привилегий, привилегированные инструкции и регистры

Архитектура RISC-V предусматривает три режима работы, которым соответствуют различные уровни привилегий.

- ***Машинный режим (M)*** *— режим, с высшим уровнем привилегий, допускает полный доступ ко всем инструкциям и регистрам.* Предназначен, например, для начальной настройки ЦПУ (компьютера),

работы начального загрузчика ОС, выполнения встроенного ПО, исполнения кода обработчика некоторых прерываний и т. д.

- ***Режим супервизора (S)*** *— режим с промежуточным уровнем привилегий, предназначенный для исполнения ядра операционных систем.*
- ***Пользовательский режим (U)*** *— режим с низшим уровнем привилегий, предназначенный для исполнения кода приложений пользовательского пространства.*

Согласно спецификации RISC-V обязательным к реализации на конкретных моделях процессоров является только машинный режим. Процессоры, в которых реализован только машинный режим, могут использоваться, например, в небольших встроенных системах, процессоры, в которых реализован только машинный и пользовательский режимы — в защищенных встроенных системах. Для запуска ОС общего назначения (например, UNIX-подобных), требуется реализация всех трех режимов.

В привилегированных режимах доступны для чтения и изменения регистры специального назначения — ***регистры контроля и статуса (Control and Status Registers***, ***CSR)***. Регистры, имена которых начинаются с m доступны из машинного режима, регистры, имена которых начинаются с s — из режима супервизора. Например, в регистре mstatus — регистре машинного уровня — с помощью битовой маски указывается в т. ч. разрешение и запрет прерываний в соответствующих битовых полях, регистр sstatus — аналогичный регистр режима супервизора, имеет меньшее количество доступных битов.

Регистры контроля и статуса могут быть прочитаны и изменены только с помощью специальных машинных инструкций — csrr (*Control and Status Register Read* — чтение регистра контроля и статуса), csrw (C*ontrol and Status Register Write* — запись регистра контроля и статуса) и другие. Например,

```
csrr a2, mstatus; // сохраняет значение регистра mstatus
                  // в регистре a2
csrw mstatus, t1; // записывает значение регистра t1
                  // в регистр mstatus
```

Более полный перечень и синтаксис инструкций следующий. В описании синтаксиса под *csr* всегда будет пониматься регистр контроля и статуса, в то время как использовавшийся ранее и используемые здесь обозначения *rs* (регистр-источник), *rd* (регистр-назначение) — только для регистров общего назначения.

Инструкция	Синтаксис	Действие

CSR Read	`csrr` *rd*, *csr*	Прочитать значение регистра *csr* в регистр *rd*
CSR Write	`csrw` *csr*, *rs*	Записать значение регистра *rd* в регистр *csr*
CSR Read and Write	`csrrw` *rd*, *csr*, *rs*	Записать значение регистра *csr* в регистр *rd*, а значение регистра *rs* — в регистр *csr* (атомарно)
CSR Read and Set bit CSR Read and Clean bit	`csrrs` *rd*, *csr*, *rs* `csrrc` *rd*, *csr*, *rs*	Записать текущее значение регистра *csr* в регистр *rd*, значение регистра *csr* изменить в соответствие с битовой маской регистра *rs* — установленные в *rd* биты в значении регистра *csr* устанавливаются (csrrs) / зануляются (csrrc) атомарно. Если в качестве *rs* указать x0 (zero), то регистр *csr* остается неизменным.
CSR Read and Write Immediate CSR Read and Set bit Immediate CSR Read and Set bit Immediate	`csrrwi` *rd*, *csr*, *imm* `csrrsi` *rd*, *csr*, *imm* `csrrci` *rd*, *csr*, *imm*	Аналогично инструкциям csrrw, csrrs и csrrc соответственно, но используя непосредственное значение в качестве записываемого значения / битовой маски.

Попытка осуществления операции с регистром контроля и статуса, недоступного для данного уровня привилегий, приведет к генерации соответствующей ошибки (исключительной ситуации).

Изначально компьютер запускается в машинном режиме. ОС имеет возможность понизить уровень привилегий описанным в конце данного пункта способом. Срабатывание прерывания, исключения, вызов инструкции системного вызова (ecall) и т. п. ситуации приводят к повышению уровня привилегий до машинного или супервизора, в зависимости от настройки процессора. Для определенности рассмотрим случай передачи управления в машинный режим. При таком переходе изменяются соответствующие биты регистра mstatus, в т. ч. в данном регистре в этих битах сохраняется предыдущий уровень привилегий. Также текущее значение регистра pc (адрес инструкции, на которой сработало прерывание, исключение или иная ловушка), автоматически сохраняется в регистре mepc, т. е. этот регистр будет содержать точку возврата из обработчика.

Это почти аналогично сохранению точки возврата при вызове функции в регистре ra, но для системного вызова и возврата из обработчика используются другие машинные инструкции и регистры, т. к. требуется обеспечить изо-

ляцию пространства ядра и пользовательского пространства. Кроме того, в качестве точки возврата в mepc сохраняется адрес инструкции на которой произошла ловушка, а не следующей за ней. Это необходимо в том числе для того, чтобы при устранении операционной системой нефатальной ошибки инструкция могла быть выполнена повторно (при системном вызове регистр mepc нужно инкрементировать на длину инструкции). При передаче управления в режим супервизора для этих целей используются соответственно регистры sstatus и sepc.

Для возврата из обработчика прерывания или исключительной ситуации следует вызвать инструкцию `mret` (если обработчик исполняется в машинном режиме), или `sret` (если обработчик исполняется в режиме супервизора). Инструкции не имеют параметров. Они восстанавливают предыдущий уровень привилегий по сохраненному в регистре mstatus (mret) или sstatus (sret) и передают управление по адресу, сохраненному в регистре mepc или sepc соответственно. Специальной инструкции для явного понижения привилегии, например, при старте ядра ОС из машинного режима или при старте пользовательского процесса ядром ОС нет. Для этого нужно использовать инструкции mret и sret соответственно, предварительно корректно выставив адрес передачи управления в mepc/sepc и «предыдущий» (т. е. целевой) уровень привилегий в mstatus/status.

Ловушки, прерывания и исключения; переключение контекста потоков и процессов

В терминологии RISC-V используются понятия ***исключения***, ***прерывания*** и ***ловушки***. При этом под ***прерыванием*** *понимается программное или аппаратное прерывание или прерывание от таймера, под* ***исключением*** *— возникновение ошибки вследствие исполнения инструкции*. Такой ошибкой может быть, например, ***некорректная инструкция*** (битовая последовательность, которая не является корректной машинной инструкцией), ***некорректный адрес*** (последовательность, которая не может быть интерпретирована как физический или виртуальный адрес в данном адресном пространстве), ***ошибка выравнивания*** (если платформа требует выравнивание начала инструкции или данных по машинным словам), а также собственно инструкция ***системного вызова*** ecall или ***точки прерывания отладчика*** ebreak.

Под понятием ***ловушки*** объединяются обе эти ситуации — и прерывания, и исключения. При этом используется термин ***переключения при возникновении ловушки*** *— автоматической передачи процессором управления по заданному (в специальных регистрах контроля и статуса —* mtvec *и* stvec,

описанных на стр. 99) адресу вследствие возникновения ловушки (прерывания и исключения) с повышением привилегии.

При срабатывании ловушки процессор автоматически переходит в соответствующий режим привилегий, сохраняет адрес точки возврата и передает управление по соответствующему адресу (изменяет регистр pc). При этом процессор не изменяет и не сохраняет регистры общего назначения. Таким образом, в тот момент, когда операционная система, получив управление от пользовательского процесса в результате прерывания, исключения или системного вызова, все регистры по-прежнему содержат данные потока пользовательского процесса, указатель стека указывает на стек пользовательского пространства и т. д. В том числе при системном вызове операционная система получит значения параметров вызова и номер системного вызова в регистрах a0 — a7. Таким образом, для корректной работы операционная система должна сохранить значения всех этих регистров, например, в стеке. Кроме того, так как реальные системы обычно используют отдельный стек ядра для потоков пользовательского процесса и собственную таблицу страниц ядра, регистры sp и регистр контроля и статуса, ответственный за работу страничной адресации — satp (описан на стр. 103) — должны быть выставлены соответственно. После этого система может выполнить необходимый код обработки ловушки (в т. ч. системного вызова), а затем восстановить значения регистров.

Для переключения контекста потока и/или процесса, которое обычно также возникает при обработке ловушки (например, при прерывании по таймеру или блокирующем системном вызове), также требуется сохранить текущее значение регистров исполняемого потока, в том числе значение регистра sepc, содержащего адрес инструкции, на которой исполнение процесса прерывается. После этого следует загрузить значения регистров нового потока, в т. ч. sepc и, при переключении контекста процесса, satp, и выполнить инструкцию sret для передачи ему управления.

Следует также обратить внимание на привилегированную инструкцию wfi (Wait For Interrupt) — ожидание прерывания (события). Данная инструкция приостанавливает выполнение инструкций до момента возникновения прерывания. После возникновения прерывания, его обработки и возврата из обработчика начинает исполняться следующая инструкция. Таким образом, процессор переводится в режим пониженного энергопотребления, например, в случае простоя из-за отсутствия потоков выполнения, ожидающих процессорное время, а потом выводится из него, например, при прерывании таймера. Реальная реализация процессора может не поддерживать понижение

энергопотребления и режим ожидания, реализуя данную операцию как nop (ничего не делать).

Регистры контроля и статуса

Имена регистров контроля и статуса машинного режима начинаются с символа m, супервизора — с символа s. Полный список регистров не регламентирован и может зависеть от реализации процессора. Спецификацией допускается также реализация регистров контроля и статуса пользовательского режима, имена которых начинается с u (в т. ч. допускается и обработка прерываний в пользовательском режиме с помощью регистра ustatus и инструкции uret). *Строго говоря, инструкции чтения и изменения регистров контроля и статуса не являются привилегированными, таковыми являются сами регистры, точнее их отдельные биты: для них регулируется право на чтение и изменение в соответствии с режимом*.

Приведем примеры основных регистров, доступных в машинном режиме.

- Регистр mvendorid содержит идентификатор производителя процессора, mimpid — идентификатор реализации (типа, модели) процессора данного производителя, marchid — идентификатор архитектуры (соответствие архитектуры версии стандарта), mhartid — идентификатор конкретного ядра процессора на многопроцессорной системе. Доступны только для чтения.
- Регистр misa содержит информацию о поддерживаемых расширениях в соответствии с битовой маской (1 означает поддержку расширения, 0 — отсутствие поддержки), а также информацию о разрядности процессора и длине регистров машинного режима (MXLEN). Стандарт RISC-V допускает реализацию процессоров без поддержки данного регистра, тогда его значение всегда равно 0. Может быть доступен для записи с целью отключения и включения расширений, а также смены разрядности процессора. Смысл некоторых битов представлен в таблице

Бит №	**Символ**	**Пояснение**
0	A	Атомарное расширение
2	C	Сжатые инструкции
3	D	Поддержка чисел с плавающей точкой двойной точности

5	F	Поддержка чисел с плавающей точкой одинарной точности
8	I	Базовые целочисленные операции
12	M	Целочисленное умножение и деление
18	S	Наличие поддержки режима супервизора
20	U	Наличие поддержки пользовательского режима
MXLEN - 1	MXL	Значение MXLEN. Соответствие: 00 — 32 бита, 10 — 64 битов, 11 — 128 битов.
MXLEN - 2	MXL	

Значение MXLEN кодируется в двух старших битах, чей номер зависит от MXLEN, поэтому требуется аккуратное их прочтение, например с помощью битового сдвига до зануления регистра.

- Регистр mstatus содержит информацию о статусе и управлением прерываниями. Смысл некоторых битов регистра представлен в таблице

Бит №	**Символ**	**Пояснение**
0	UIE	Разрешены прерывания в пользовательском режиме
1	SIE	Разрешены прерывания в режиме супервизора
3	MIE	Разрешены прерывания в машинном режиме
8	SPP	Уровень привилегий до вызова прерывания в режим супервизора (процессор сохраняет здесь уровень привилегий при срабатывании ловушки и устанавливает заданный здесь уровень при вызове sret): 0 — пользовательский режим, 1 — режим супервизора
12-11	MPP	Уровень привилегий до вызова прерывания в машинный режим (процессор сохраняет здесь уровень привилегий при срабатывании ловушки и устанавливает заданный здесь уровень при вызове mret): 00 — пользовательский режим, 01 — режим супервизора, 11 — машинный режим

Т. к. у каждого ядра этот (как и любой другой) регистр свой, прерывания могут быть запрещены и разрешены на отдельном ядре.

- Регистр mepc содержит адрес инструкции, вызвавшей исключение или на которой произошло прерывание (сохраненное значение регистра pc), т. е. адрес, куда будет передано управление инструкцией mret. Данный регистр может быть изменен и явным образом для передачи управления по заданному адресу с понижением привилегий до уровня, заданного в битах MPP регистра mstatus.

- В регистр mcause процессор записывает причину, вызвавшую исключение или прерывание. Старший бит регистра содержит 0, если причина — исключение (ошибка, системный вызов) или 1, если причина — аппаратное прерывание, остальные биты содержат код причины.

Прерывание	**Код причины**	**Пояснение**
1	1	Программное прерывание в режим супервизора
1	3	Программное прерывание в машинный режим
1	5	Прерывание таймера в режим супервизора
1	7	Прерывание таймера в машинный режим
1	9	Внешнее прерывание в режим супервизора
1	11	Внешнее прерывание в машинный режим
1	≥ 16	Платформозависимые коды
0	0	Некорректное выравнивание адреса инструкции
0	1	Ошибка адреса инструкции
0	2	Некорректная инструкция
0	3	Точка прерывания отладчика (инструкция ebreak)
0	4	Некорректное выравнивание адреса при чтении из памяти
0	5	Ошибка адреса чтения из памяти
0	6	Некорректное выравнивание адреса при записи в память
0	7	Ошибка адреса записи в память
0	8	Системный вызов из пользовательского режима (инструкция ecall)
0	9	Системный вызов из режима супервизора (инструкция ecall)
0	11	Системный вызов из машинного режима (инструкция ecall)
0	12	Ошибка страницы при чтении инструкции, подлежащей исполнению (трансляция адреса)
0	13	Ошибка страницы при чтении из памяти (трансляция адреса)
0	15	Ошибка страницы при записи в память (трансляция адреса)

- В регистр mtval процессор записывает адрес инструкции, вызвавшей исключение.
- Регистр mtvec используется для указания адреса обработчика ловушек (ошибок и прерываний). Его значение должно быть записано операционной системой для корректной передачи управления обработчику при возникновении такой ситуации. Младшие два бита содержат режим определения адреса обработчика: ***прямой*** (00) или ***векторизированный*** (01). В прямом режиме оставшиеся биты содержат адрес обработчика, который вызывается вне зависимости от причины. В векторизированном режиме адрес обработчика вычисляется как значение регистра плюс значение причины (из регистра mcause), умноженное на разрядность адреса. Именно этот (прямой или вычисленный) адрес будет записан в регистр pc при возникновении ловушки. Таким образом, в прямом режиме используется единый обработчик для всех исключений и прерываний, а то время как в векторном режиме нужно устанавливать отдельный адрес для каждой причины.
- Регистры medeleg и mideleg позволяют делегировать обработку исключений и прерываний соответственно в режимы с более низкой привилегией. По умолчанию все ловушки переводят процессор в машинный режим, но это может быть избыточно. В этом случае система должна выставить соответствующие значения битов этих регистров. Регистры mip и mie в свою очередь содержат биты, разрешающие или запрещающие сами прерывания и исключения соответственно, а также их делегирование. Биты всех четырех регистров соответствуют типам причины, указываемым в mcause (код причины *i* соответствует номеру бита регистра). Таким образом, чтобы прерывание *i* привело к переходу в машинный режим из более низкого уровня привилегий (в т. ч. пользовательского), нужно соблюдение всех условий: бит MIE регистра mstatus должен быть установлен, бит *i* должен быть установлен в регистре mip, бит *i* не должен быть установлен в регистре mideleg.
- Регистры pmp0cfg — pmp63cfg и pmpaddr0 — pmpaddr63 позволяют установить защиту физической памяти для корректной работы режима страничной адресации и трансляции адреса, например, установить дополнительную защиту от доступа к таблицам страниц и страница ядра.

В режиме супервизора доступны, в частности, регистры sstatus, scause, stval и stvec, имеющие аналогичный смысл (и порядок битов), что и одноименные регистры машинного режима. Регистр sstatus может рассматриваться как подмножество регистра mstatus. Кроме того, режим супервизора позволяет включить режим страничной адресации с помощью регистра satp: если значение данного регистра равно 0, то страничная адресация отключена и используются физические адреса, установка ненулевого значения регистра включает страничную адресацию, причем значение регистра используется как адрес таблицы страниц. В машинном режиме страничная трансляция адреса не поддерживается.

Трансляция адреса в RISC-V

Адрес таблицы страниц хранится в регистре satp. Изменение этого регистра осуществляется с помощью привилегированной инструкции csrw, чтение — csrr, например

```
csrr a2, satp;  // сохраняет значение регистра satp
                // в регистре a2
csrw satp, t1;  // записывает значение регистра t1
                // в регистр satp
```

Размер страницы обычно составляет 4 килобайта (но может быть изменен). В зависимости от реализации и конфигурации виртуальный (и физический) адрес в RV64I может быть 39-битным (SV39), что позволяет, соответственно, адресовать до 512 гигабайтов виртуального адресного пространства или 48-битным (SV48), адресующим до 256 терабайтов. Возможно расширение до SV57 и выше.

В SV39 таблица страниц трехуровневая. Виртуальный адрес состоит из трех 9-битных индексов записей в таблице страниц каждого уровня (позволяет адресовать 512 записей) и 12-битного смещение внутри страницы (позволяющее адресовать 4096 байтов страницы). Каждая ***запись в таблице страниц*** (***Page Table Entry***, PTE) содержит 44-битный ***номер физической страницы*** (***Physical Page Number***, PPN), однозначно определяющий адрес ее начала в физической памяти, и 9 битов флагов. Каждая PTE занимает 8 байтов, а таблица страниц каждого уровня занимает одну страницу физической памяти и содержит 512 PTE. Многоуровневые таблицы страниц позволяют экономить память, так как не требуют размещать в памяти таблицу всех возможных страниц: достаточно создать несколько небольших таблиц для реально выделенных участков памяти.

При трансляции адреса процессор находит в таблице страниц второго уровня, расположенной в физической странице по адресу, записанному в регистре satp, запись с индексом L2 (старшей частью виртуального адреса), извлекает из неё PPN, содержащий номер физической страницы с таблицей страниц первого уровня, находит в ней PTE с индексом L1, извлекает из нее PPN, содержащий номер физической страницы с таблицей страниц нулевого уровня, находит в ней PTE с индексом L0 и обращается к адресу физической страницы с номером PPN из данной таблицы со смещением, записанной в младшей части виртуального адреса. Режим SV48 использует четырехуровневые таблицы страниц, SV57 — пятиуровневые, работающие по аналогичному принципу.

Флаги страницы следующие:

Бит	Название	Назначение
0	D (Dirty)	Страница была изменена
1	A (Accessed)	К странице был доступ
2	G (Global)	Использование страницы несколькими задачами
3	U (User)	Доступность страницы в пользовательском режиме
4	E (Executable)	Страница доступна для исполнения
5	W (Writable)	Страница доступна для записи
6	R (Readable)	Страница доступна для чтения
7	V (Valid)	Свидетельство корректной страницы
8-9	(Reserved)	Зарезервировано для нужд ОС

Биты D и A автоматически выставляются процессором в 1 в момент изменения (записи) и любого обращения (чтения и записи) данных страницы соответственно. Их можно изменить программно средствами ОС, в частности сбросить в 0. Это позволяет ОС проверить, были ли изменения и обращения к странице со стороны процесса с момента последнего сброса. Все остальные биты могут изменяться только программно операционной системой.

Схема виртуального адреса, таблицы страниц и трансляции адреса, следующая.

Для создания виртуального адресного пространства процесса операционная система должна для каждого процесса

- выделить по крайней мере 3 (для SV39) страницы для хранения трех уровней таблицы страниц, записать в регистр satp физический адрес первой из них;
- в таблице страниц второго уровня сделать запись с физическим адресом таблицы страниц первого уровня, в таблице страниц первого уровня сделать запись с физическим адресом таблицы страниц нулевого уровня, в таблице страниц нулевого уровня сделать запись с адресом физической страницы данных процесса;
- для всех неиспользуемых записей установить соответствующий бит V в 0 (или просто заполнить нулями неиспользуемые записи), чтобы исключить трансляцию адреса, если индекс PPN в виртуальном адресе указывает на недействительную запись — процессор должен сгенерировать ошибку отсутствия страницы;

- выделить необходимое количество физических страниц для хранения данных процесса;
- в таблице страниц последнего уровня сделать записи о каждой такой странице, а при их нехватке выделить дополнительную страницу последнего уровня, предпоследнего уровня и т. д.

Следует обратить внимание, что переполнение смещения (инкремент адреса при смещении равном 4095) приведет к увеличению на 1 значения индекса L0 и принятию значения 0 смещением, т. е. к переходу к следующей записи в таблице страниц нулевого уровня, к началу следующей страницы. Если же этот индекс был 511, то он также переполнится — примет значение 0, но увеличится индекс L1, т. е. будет осуществлен переход к нулевой записи следующей таблице страниц первого уровня и т. д. Правильное построение таблицы страниц операционной системой позволяет создавать непрерывные в виртуальном адресном пространстве сегменты данных и стека, стековый сегмент, выделять память из кучи, при этом располагая физические страницы в памяти произвольным образом, не заботясь о фрагментации. Возможна в том числе реализация расширения сегмента данных процесса путем добавки новых страниц в конец, а также сегмента стека путем добавки страниц в начало (при наличии резерва в таблице страниц).

Ввод и вывод

Контроллеры внешних (по отношению к центральному процессору) устройств — контроллеры устройств ввода/вывода, контроллеры прерываний, контроллеры шины и иных устройств — снабжаются собственными ***управляющими регистрами*** (которые не следует путать с регистрами процессора, установленного на контроллере). Операции ввода/вывода и настройки работы устройства осуществляются путем чтения и записи значений этих регистры.

Операции ввода/вывода должны быть разрешены только в привилегированном режиме. В ряде архитектур им соответствуют специальные привилегированные инструкции. В таких случаях каждому внешнему устройству, точнее регистру его контроллера, сопоставляется ***порт ввода/вывода***, идентифицируемый по уникальному номеру. Привилегированные инструкции ввода/вывода представляют собой инструкции записи и чтения информации из порта в регистр или в память. Такой ввод/вывод называется ***вводом/выводом посредством отображение портов*** (PMIO, *Port-Mapped Input/Output*).

Альтернативный подход заключается в *отображении каждого регистра контроллера (или порта) устройства ввода/вывода в адресное пространство физической оперативной памяти*. Этот подход носит название ***ввод/вывод по-***

средством отображение в память (MMIO, *Memory-Mapped Input/Output*). При таком подходе в адресном пространстве физической памяти выделяются обычно непересекающиеся диапазоны адресов для собственно физической памяти и для портов устройств ввода/вывода. Для операций ввода/вывода не требуется отдельных машинных инструкций, так как они обеспечиваются обычными инструкциями загрузки в регистр и сохранения данных в память. Привилегированность операций ввода/вывода обеспечивается тем, что процессы пользовательского режима не имеют (в норме) доступа к физическим адресам (ОС может его предоставить, отобразив некоторые виртуальные адреса на соответствующие физические, тем самым тонко настроив права процессов на работу с внешними устройствами, но обычно такой подход не применяется).

В архитектуре RISC-V используется ввод/вывод посредством отображение в память (MMIO). Конкретные адреса конкретных устройств в памяти, их регистры и работа с ними специфична для аппаратной платформы и этих устройств, поэтому их описание лежит за рамками данного пособия.

Глава 3. Операционная система xv6

Глава содержит описание ОС xv6: структуры исходного кода, интерфейса системных вызовов, интерфейса прикладного программирования и программирования в пространстве ядра, компонентов управления процессами и виртуальной памятью, механизмов синхронизации, файловой системы. Операционная система xv6 разрабатывается в MIT с 2006 года как простая учебная операционная система, то есть она ориентирована на изучения курса по проектированию операционных систем. Система позиционируется как современная открытая реализация на архитектуре RISC-V ОС Unix Version 6, выпускавшейся для компьютеров PDP11/40. В операционной системе реализованы простые (сокращенные и упрощенные) интерфейсы системных вызовов и средств командной строки, похожие на аналогичные в POSIX. При этом следует отметить, что

- в xv6 поддерживается многозадачность и одновременная многопоточность (параллельная работа нескольких процессоров);
- в xv6 не поддерживаются многопоточные процессы, т. е. все процессы имеют ровно один поток выполнения, сам термин «поток», соответственно, не используется — рассматривается только управление собственно процессами;

- в xv6 не поддерживается многопользовательский режим и контроль прав доступа, все процессы запускаются с полными административными привилегиями;
- ОС xv6 рассчитана на запуск на виртуальной машине QEMU и не имеет драйверов для работы на реальной аппаратуре, равно как и развитого интерфейса для их написания;
- Исходный код xv6 написан на языке Си и, там где это необходимо, на Ассемблере, и в обоих случаях компилируется стандартным компилятором GCC, но при этом не компонуется со стандартной библиотекой языка Си (glibc или аналогичной), используется ее упрощенная (сокращенная) реализация внутри ОС.

В виду того, что в xv6 не поддерживаются многопоточные процессы и отсутствуют какие-либо отдельные структуры ядра, ответственных за потоки, в данной главе не акцентируется внимание на существовании отдельной сущности «поток выполнения», под «процессом» понимается как собственно процесс, так и его единственный и всегда существующий поток выполнения.

Более детальное описание ОС xv6 можно получить в руководстве от разработчиков ОС (проводится в Массачусетском технологическом институте), а также читая исходный код операционной системы.

1. Russ Cox, Frans Kaashoek and Robert Morris «xv6: a simple, Unix-like teaching operating system». Распространяется свободно, в т. ч. в виде исходного кода по адресу `https://github.com/mit-pdos/xv6-book` (дата доступа 01.02.2024). PDF-версия обычно доступна на странице курса MIT по проектированию операционных систем — Operating System Engineering. (последняя на момент написания данного пособия реализация курса была осенью 2023 года, PDF-версия доступна по адресу `https://pdos.csail.mit.edu/6.828/2023/xv6/book-riscv-rev3.pdf`, дата доступа 01.02.2024).
2. Исходный код ОС xv6 распространяется свободно и доступен по адресу `https://github.com/mit-pdos/xv6-riscv` (дата доступа 01.02.2024).

Полный исходный код xv6 (на момент написания пособия) содержит около 12000 строк кода. Код написан с большим количеством комментариев и с приоритетом на понятность, а не эффективность, что делает его простым для изучения. Чтение исходного кода рекомендуется как при изучении данной главы, так и собственно при решении задач из главы 4.

В **генеральном каталоге** репозитория находятся файлы LICENSE и README, а также Makefile, используемый для сборки и запуска ОС, и шаблонные и настоечные файлы для Git и gdb.

Каталог **`kernel`** содержит код ядра ОС, каталог **`user`** — код приложений пользовательского пространства ОС (заголовочные файлы языка Си с расширением `.h`, файлы исходного кода на Си и на Ассемблере с расширениями .c и .S соответственно). Оба каталога также содержат файлы `kernel.ld` и `user.ld`, содержащие сценарии компоновщика исполняемых файлов под архитектуру RISC-V. Исходные коды данных каталогов компилируются под архитектуру RISC-V и исполняются на виртуальной машине.

Каталог **`mkfs`** содержит исходный код `mkfs.c` программы mkfs для создания образа файловой системы xv6. Данный файл, в отличие от ядра и утилит пользовательского пространства, компилируется под хост-платформу, на которой исполняется виртуальная машина QEMU с xv6 и запускается на ней как часть процедуры сборки ОС. Получаемый файл — `fs.img` — записывается в генеральный каталог репозитория. Именно он используется в качестве виртуального жесткого диска для виртуальной машины.

Данная глава ориентирована на описание концепций и структуры кода ОС, а не на детальное техническое описание реализаций задач операционной системы и их теоретическое обоснование. Последнюю информацию следует искать в официальном описании и с помощью чтения исходного кода системы. Также, в случае обновления xv6 разработчиками, содержание исходного кода может измениться. Рекомендуется всегда использовать новейшую версию ОС и актуальное описание.

Дальнейшее изучение реальных операционных систем — в частности, системного программирования в ОС на базе Linux (в т. ч. средствами POSIX) и устройства ядра операционной системы Linux можно проводить с использованием следующей литературы.

1. Роберт Лав «Linux. Системное программирование».
2. Роберт Лав «Ядро Linux: описание процесса разработки».
3. Эндрю Таненбаум, Альберт Вудхалл «Операционные системы: разработка и реализация».
4. John Madieu «Linux Device Driver Developement».
5. John Madieu «Mastering Linux Device Driver Developement».
6. Sreekrishnan Venkateswaran «Essential Linux Device Drivers».

§3.1. Пользовательское пространство xv6

Доступ к API пользовательского пространства

API пользовательского пространства xv6 представлен системными вызовами и библиотекой функций языка Си. Прототипы библиотечных функций и оберточных Си-функций системных вызовов расположены в файле `user/user.h`. Код библиотечных функций расположен в файлах `user/ulib.c`, `user/umalloc.c`, `user/printf.c`.

Строго говоря, xv6 содержит *только* прототипы оберточных Си-функций системных вызовов, код их тела представлен исключительно на Ассемблере — компоновщик связывает имя Си-функции с ассемблерной меткой. Ассемблерный исходный код системных вызовов генерируется автоматически с помощью perl-скрипта `user/user.pl` и записывается в файл `user.S`.

В библиотеке Си xv6 не вводятся разнообразные стандартные типы данных языка, такие как `size_t` или `FILE*` (используются только целочисленные — типа `int` — дескрипторы открытых файлов, буферизация в пользовательском пространстве не реализована). Вводятся псевдонимы для беззнаковых вариантов `int`, `short` и `char` — `uint`, `ushort` и `uchar` соответственно, а также беззнаковые типы фиксированной разрядности — `uint8`, `uint16`, `uint32` и `uint64` (точный аналог `stdint.h` отсутствует).

Функции библиотеки языка Си

В xv6 представлена сокращенная (упрощенная) и несколько измененная реализация некоторых функций стандартной библиотеки языка Си.

- ***Строковые операции и операции с блоками памяти.*** Аналогичны стандартным функциями языка Си.
  - Получение длины строки, копирование строк, поиск символа в строке и сравнение строк соответственно:

```
uint  strlen(const char *s);
char *strcpy(char *s, const char *t);
char *strchr(const char *s, char c);
int   strcmp(const char *p, const char *q);
```

  - Заполнение памяти байтом (`c` — байт для записи, `n` — количество символов), сравнение блоков памяти, копирование блоков

памяти (неперекрывающиеся буферы), копирование блоков памяти (возможно перекрывающиеся буферы) соответственно:

```
void *memset(void *dst, int c, uint n);
int  memcmp(const void *s1, const void *s2, uint n);
void *memcpy(void *dst, const void *src, uint n);
void *memmove(void *vdst, const void *vsrc, int n);
```

- Преобразование десятичного текстового представления числа в двоичное типа `int` (поддерживаются только положительные числа):

```
int atoi(const char *s);
```

- ***Функции ввода/вывода.***
  - «Форматированный» вывод на консоль и в открытый для записи файл соответственно. Открытие и закрытие файла осуществляется с помощью системных вызовов, описанных на стр. 115. Буферизированный в пользовательском пространстве ввод/вывод не поддерживается. Поддерживается ограниченное количество формата полей — `%d` (десятичный, `int`), `%x` (шестнадцатеричный, `int`), `%p` (указатель), `%s` (строка `char *`). Не поддерживается собственно формат — ширина поля, точность и т. д.):

```
void printf(const char *fmt, ...);
void fprintf(int fd, const char *fmt, ...);
```

  - Ввод строки с консоли. В отличие от стандартной, «запрещенной» как небезопасной, функции `gets` содержит дополнительный параметр — ограничение числа вводимых символов (размер буфера):

```
char* gets(char *buf, int max);
```

- ***Функции выделения и освобождения памяти:***

```
void* malloc(uint nbytes);
void  free(void *ap);
```

- ***Функция получения информации о файле*** (аналогична POSIX), `n` — имя файла:

```
int stat(const char *n, struct stat *st);
```

Структура `stat`, содержащая информацию о файле, определена в заголовочном файле ядра `kernel/stat.h`. Читателю предлагается самостоятельно изучить поля данной структуры.

Системные вызовы xv6: создание и завершение процессов

В xv6 представлена сокращенная (упрощенная) и несколько измененная реализация некоторых системных вызовов POSIX.

Системный вызов

```
int fork(void);
```

создает процесс как полный клон (код, данные, стек, открытые файлы и т. д.) исходного процесса. Весь дальнейший код после этого вызова будет исполняться параллельно, в двух процессах. При этом один из них принимается родительским, второй (созданный) — дочерним по отношению к данному процессу. При этом в дочернем процессе функция вернет 0, в родительском — идентификатор дочернего процесса. При ошибке (невозможности создать процесс) возвращает отрицательное значение.

Системный вызов

```
int exec(const char *file, char **argv);
```

замещает текущий процесс кодом указанного исполняемого файла. Второй параметр функции — массив строк (указателей на строки) — передает массив параметров командной строки запускаемого файла. Последним указателем должен быть NULL (для определения количества элементов в массиве, т. е. это NULL-терминированный массив 0-терминированных строк). *Заметим, что в API* xv6 *не используется константа* NULL, *поэтому указывается просто 0, приводимый к указателю.* В случае успеха исполнение кода текущего процесса прекращается, при ошибке — продолжается.

Системный вызов

```
int exit(int status);
```

завершает процесс с указанным статусом. Как и в Linux, ОС освобождает все задействованные процессом ресурсы, в т. ч. закрывает открытые файлы, а процесс переводится в состояние «зомби» до тех пор, пока его код возврата не будет прочитан родительским процессом.

Строго говоря, исполнение этого системного вызова для завершения программы *обязательно*. Если этого не сделать, процессор *не передаст* управление операционной системе, продолжив исполнять данные, расположенные в памяти за последней инструкцией кода приложения, что является непредсказуемым поведением, так как сами данные и их наличие (момент выхода за границы адресного пространства процесса) непредсказуемы. По умолчанию компиляторы языка Си автоматически делают этот вызов в

соответствие с возвращаемым значением функции main. В xv6 это реализовано на уровне настроек компоновщика, с помощью оберточной функции _main: именно этой функции и передается управление при запуске программы. Код данной функции следующий (представлен в `ulib.c`):

```
_main()
{
  extern int main();
  main();
  exit(0);
}
```

В таком виде функция не получает возвращаемое значение функции main, поэтому он не используется как код возврата приложения (в отличие от реальных операционных систем). Для того, чтобы код возврата программы имел ненулевое значение, вызов exit должен быть сделан явно. На момент написания пособия имеется предложение о внесении исправления в код xv6, поэтому данная информация может быстро потерять актуальность.

Системный вызов

```
int getpid(void);
```

возвращает идентификатор текущего процесса (сделавшего данный вызов).

Системный вызов

```
int wait(int *status);
```

приостанавливает исполнение текущего процесса в ожидании завершения дочернего процесса (при наличии). Возвращает идентификатор завершенного процесса, если дочерних процессов нет, возвращает отрицательное значение. В `*status` записывается код возврата завершенного процесса (если передан нулевой указатель, параметр игнорируется).

Типичной схемой запуска приложения как дочернего процесса (например, таким способом оболочка может запускать программы), является следующая:

```
int pid = fork();   // Создаем дочерний процесс
if (pid < 0)        // Ошибка fork, процесс не создан
{
    // Вывод сообщения об ошибке в поток ошибок,
    // как и в POSIX его идентификатор – 2
    // 0 – поток ввода, 1 – поток вывода
    fprintf(2, "fork error\n");
}
```

```
else if (pid > 0)   // Мы в родительском процессе
{
    int status, cpid;
    cpid = wait(&status);  // Ожидаем завершения
                           // дочернего процесса
    printf("child pid = %d exit with status %d\n",
           cpid, status
          ); // Информация о завершенном процессе
             // В этом примере pid и cpid должны
             // совпадать (или cpid = -1 при ошибке)
    exit(0); // Завершение родительского процесса
}
else if (pid == 0)  // Мы в дочернем процессе
{
    char *argv[3];              // Массив аргументов
    argv[0] = "echo";           // Нулевой аргумент –
                                // имя программы
    argv[1] = "Hello, world";   // Параметры командной
    argv[2] = "from";           // строки запускаемой
    argv[3] = "parent";         // программы
    argv[4] = (void *)0;        // Или просто 0 –
                                // завершение списка
    exec("/echo", argv);        // Абсолютный путь
    fprintf(2, "exec error\n"); // Если попали сюда,
                                // файл не запущен
    exit(1);      // Завершаем дочерний процесс
                  // (с кодом ошибки)
}
```

Если в родительском процессе не выполнить вызов `wait`, то дочерний процесс останется в состоянии «зомби» до завершения родительского процесса, после чего будет передан как дочерний процесс самому первому процессу пользовательского процесса ОС. В xv6 таковым является программа init. В реальных UNIX-подобных ОС такую роль может играть программа с аналогичным названием или демон systemd. Имя init часто используется как нарицательное для программ подобного класса. Задачи этой программы — исполнение сценария запуска ОС (в т. ч. запуск служб и демонов), вызова программы входа в систему, запуска оболочки и т. д. Одной из функций этой программы также является сбор кодов возврата всех возникших «зомби»-процессов, чтобы они не остались в таком состоянии «вечно» (до перезапуска системы).

Системные вызовы xv6: ввод и вывод

Системный вызов

```
int open(const char *filename, int flags);
```

открывает файл и, в случае успеха, возвращает его файловый дескриптор (положительное, в т. ч. нулевое число) или отрицательное значение при ошибке. Флаги открытия файла представляют собой битовую маску следующих значений (определены в `kernel/fcntl.h`):

O_RDONLY	Открыть только для чтения
O_WRONLY	Открыть только для записи
O_RDWR	Открыть для чтения и записи
O_CREATE	Создать файл, если он не существует
O_TRUNC	Удалить содержимое файла, если он существует

В реальных системах поддерживается большее количество флагов. В xv6, как и предусмотрено в стандарте POSIX, изначально для каждого процесса открываются 3 файловых дескриптора: 0 — поток ввода (только чтение), 1 — поток вывода (только запись), 2 — поток ошибок (только запись). Они могут быть использованы по назначению или закрыты при необходимости. Также оболочка xv6 поддерживает стандартный синтаксис перенаправления потоков ввода и вывода (но не ошибок) в файл.

Системный вызов

```
int close(int fd);
```

закрывает ранее открытый файл по файловому дескриптору. Возвращает 0 при успехе или отрицательное значение при ошибке.

Системный вызов

```
int dup(int fd);
```

дублирует открытый файловый дескриптор. Возвращает отрицательное значение при ошибке или новый файловый дескриптор при успехе. Новый файловый дескриптор представляет собой клон исходного: будет открыт тот же самый файл точно таким же образом (с теми же флагами, в той же позиции чтения/записи). Но новый и старый дескриптор будет независимыми: файловые операции будут исполняться независимо, смещение позиции при последовательном чтении будет проходить независимо, один из дескрипторов можно закрыть независимо от другого и т. д. С помощью dup можно, например, перенаправить поток ошибок в тот же файл, что и поток вывода.

```
close(2);
dup(1);
```

Если эти две операции будут выполнены подряд, то новый открытый дескриптор — дубликат потока вывода — получит именно номер 2, т. к. номера выделяются подряд. Чтобы перенаправить стандартные потоки приложения, оболочка может поступить, например, по такой схеме (проверки ошибок опущены для ясности):

```
int pid = fork();
if (pid == 0)
{
    close(0);  // закрытие всех
    close(1);  // открытых файловых
    close(2);  // дескрипторов
    fopen ("in_file",  O_RDONLY); // открытие потока 0
    fopen ("out_file", O_WRONLY); // открытие потока 1
    dup(1); // открытие потока 2 как дубликата потока 1
    exec("command", argv);        // запуск приложения
    exit(-1);                     // выход при ошибке
}
```

Это сработает потому, что системный вызов exec *замещает код, стек и данные исполняемого процесса новым приложением, но открытые файловые дескрипторы сохраняются.* Аналогичным образом оболочка может заместить поток ввода или вывода открытым дескриптором конца канала (описаны на странице 119).

Системный вызов

```
int write(int fd, const void *buf, int size);
```

выводит `size` байтов из буфера `buf` в файл `fd`. Возвращает число выведенных байтов или отрицательное значение при ошибке. Заметим, что выведено может быть меньше байтов, чем требуется — это может свидетельствовать об ошибке записи или о временной невозможности совершить запись по иным причинам. Простейшая проверка на ошибку записи может состоять в следующей схеме.

```
int fd = open ("out_file.txt", O_WRONLY);
                               // Открытие файла
char str[] = "Hello, world\n";
size_t len = strlen (str);
int ret = write (fd, str, len); // Вывод len байт
```

```
if (ret != len)                  // Если не вывелись все:
    fprintf(2, "Write error");   // ошибка
```

Более аккуратный подход состоит в циклических попытках вывести остаток буфера до ошибки или вывода всех данных:

```
char *buf = str;                 // Буфер вывода
while (len > 0)                  // Пока остались данные
{
    int ret = write (fd, str, len);
    if (ret < 0) break;          // Точно ошибка
    len -= ret;                  // Осталось вывести
    buf += ret;                  // С этой позиции
}
if (ret < 0)                     // Если была ошибка
    fprintf(2, "Write error"); // сообщение о ней
```

По окончании вывода также требуется закрыть файл и проверить, что это прошло успешно:

```
if (close(fd) < 0)
    fprintf(2, "Write error");
```

В противном случае нельзя гарантировать, что все данные были отправлены в файл. Впрочем, даже при успешном закрытии файла для записи остается риск потери данных при аппаратном или системном сбое, если ОС или устройство хранения данных не успеет синхронизировать внутренний кеш или буфер. Но проверка успеха закрытия файла при записи позволяет выявить хотя бы часть возможных ошибок.

Системный вызов

```
int read(int fd, void *buf, int size);
```

прочитывает не более `size` байтов в буфер `buf` из открытого файла `fd`, возвращает число фактически прочитанных байтов при успехе. Возврат 0 соответствует достижению конца файла (закрытию потока ввода и т. п.), отрицательное значение — ошибке. Как и в реальных системах, *даже если в файле (потоке ввода и т. д.) еще осталось по крайней мере* `size` *байтов данных, система не гарантирует, что они все будут прочитаны. Гарантируется, что при наличии данных будет прочитан по крайней мере 1 байт.* Реальные системы стараются оптимизировать процессы чтения/записи, поэтому прочтение однобайтовыми кусками маловероятно, но и не гарантируется получение `size` данных при наличии. Более того, если часть данных уже поступила с устройства ввода, а остальные нужно ждать, система вполне вероятно вернет часть данных. Но, *если конец файла не достигнут, а данные еще не*

поступили, процесс блокируется в ожидании данных, т. е. read *вернет что-либо только тогда, когда данные поступят или будет получена информация, что данных больше нет. Аналогично, вызов* write *может заблокировать процесс, если вывод данных в данный момент невозможен, например, из-за того, что требуется дождаться освобождения буфера устройства или канала.* Таким образом, read и write являются классическими примерами блокирующих системных вызовов.

Типичная схема прочтения файла (потока данных) с помощью read выглядит следующим образом:

```
#define BUF_SIZE 1024         // Размер буфера
int fd = open ("in_file.txt", O_RDONLY);
                              // Открытие файла
if (fd < 0)                   // Ошибка, файл не открыт
{
    fprintf(2, "Open error\n");
}
else                          // Файл открыт успешно
{
    char buf[BUF_SIZE];       // Буфер
    int r;
    while ( (r = read(fd, buf, BUF_SIZE)) > 0)
                              // Пока поступают данные
    {
        write(1, buf, r); // Поступило r байтов,
                              // вывод их на экран
                              // или обработка иным способом
                              // (накопление в памяти и др.)
    }
    if (r == 0) // Файл прочитан успешно
        printf ("Success");
    else       // Ошибка чтения
        fprintf (2, "Read error");
    close(fd); // Закрытие файла: проверка на ошибку
               // закрытия возможна, но если файл уже
               // прочитан до конца или до ошибки,
               // может считаться избыточной
}
```

Заметим, что xv6 *не поддерживает коды ошибок ввода/вывода и иных, не записывает их в переменную* errno*, подобно реальным ОС. Однако проверка факта возникновения ошибки возможна и необходима практически во всех случаях.*

Системные вызовы xv6: межпроцессное взаимодействие

Процессы изолированы между собой на уровне виртуального адресного пространства, поэтому не могут обмениваться данными просто используя общие адреса в памяти (одинаковые указатели). Это относится также и к паре из дочернего и родительского процесса, которые были созданы с помощью fork, и разделяют общий код (содержимое текстового сегмента): после клонирования у них одноименные переменные (или метки на уровне ассемблерного кода) и используемые указатели — виртуальные адреса — одинаковые, но это изолированные процессы и эти виртуальные адреса в общем случае должны отображаться на различные физические. Таким образом, значения данных (одноименных переменных) в двух процессах будут меняться независимо.

Для обмена информации между процессами требуются механизмы межпроцессного взаимодействия. В xv6 представлены сигналы и каналы, но реализована только отправка сигнала принудительного завершения процесса, а также представлены только ***анонимные каналы***, позволяющие взаимодействовать родительскому процессу и дочернему процессу, созданному с помощью fork.

```
// Принудительно завершает процесс с идентификатором pid
int kill(int pid);
// Открывает канал
int pipe(int *fds);
```

На уровне API канал представляет собой два файловых дескриптора для одного и того же файла (в данном случае анонимный канал не представлен каким-либо файлом в ФС, но это не меняет ситуации). Один дескриптор открыт для чтения с начала файла, другой — для записи в конец файла. Таким образом, через канал реализуется передача очереди байтов данных (FIFO, First In — First Out, первый пришел — первый прочитан). Аргументом функции является массив их двух элементов типа int — два дескриптора, возвращается 0 при успехе и отрицательное значение при ошибке. Типичная схема использования следующая:

```
int pfd[2];                 // Дескрипторы канала
char str = "Hello, world"; // Передаваемая строка
pid_t pid;
if (pipe(pfd) < 0)          // Создание канала
{                           // Если ошибка
 // …                           Обработка ошибки
}
```

```
pid = fork();
if (pid < 0)
{                           // Если ошибка fork
 // …                           Обработка ошибки
}
else if (pid == 0)
{   // Чтение из канала на стороне потомка
    close(pfd[1]);      // Закрытие конца для записи
    char buf[BUF_SIZE]; // Буфер чтения
    int len;            // Количество прочитанных байтов
    while ((len = read(pfd[0], &buf, BUF_SIZE)) > 0)
    write(1, &buf, len);   // Потребление данных
    if (len < 0)
    {
      //…                     Обработка ошибки чтения
    }
    close(pfd[0]);  // Закрытие конца для чтения
 exit(0);
}
else
{   // Запись в канал на стороне родителя
    close(pfd[0]);      // Закрытие конца для чтения
    int len = strlen(str);
    if (len != write(pipefd[1], str, len))
    {
     // Отправка в канал строки
     // и обработка ошибки записи
     // (например, обрыв канала)
    }
    int ret = close(pipefd[1]);
                        // Закрытие конца для записи –
                        // На стороне потомка будет
                        // получен конец файла (EOF)
    if (ret < 0)        // Ошибка закрытия канала
    {
      // …              Обработка ошибки закрытия канала
    }
    else
        wait((int *) 0); // Ожидание потомка
                         // (игнорируется код возврата)
    exit(0);
}
```

Системный вызов fork полностью дублирует процесс, включая дескрипторы открытых файлов, в т. ч. каналов. Сам канал — структура ядра — остается в единственном экземпляре. Дочерний процесс будет иметь те же данные и видеть эти дескрипторы под теми же номерами, но это другой процесс, и у него свое пространство дескрипторов открытых файлов в ядре, поэтому их закрытие в дочернем процессе не повлияет на родительский и наоборот. Канал останется существующим, пока есть хотя бы один процесс, у которого открыт хотя бы один конец канала.

Важно обратить внимание, что в этом примере если родительский процесс не выполнит закрытие конца для записи, то дочерний процесс окажется заблокированным в ожидании данных от родителя, а родительский — в ожидании завершения дочернего процесса, т. е. они оба окажутся в бесконечном взаимном ожидании друг-друга (состоянии ***взаимоблокировки***).

В ядре xv6 каналы реализованы посредством ограниченного кольцевого буфера, поэтому при записи в заполненный канал также возможна блокировка в ожидании чтения из канала. Поэтому важно обеспечивать наличие именно двух параллельно работающих процессов, использующих канал. Канал не может быть использован для хранения данных и двунаправленной передачи информации.

В реальных операционных системах принципы и особенности использования каналов аналогичны.

Системные вызовы xv6: файловые операции

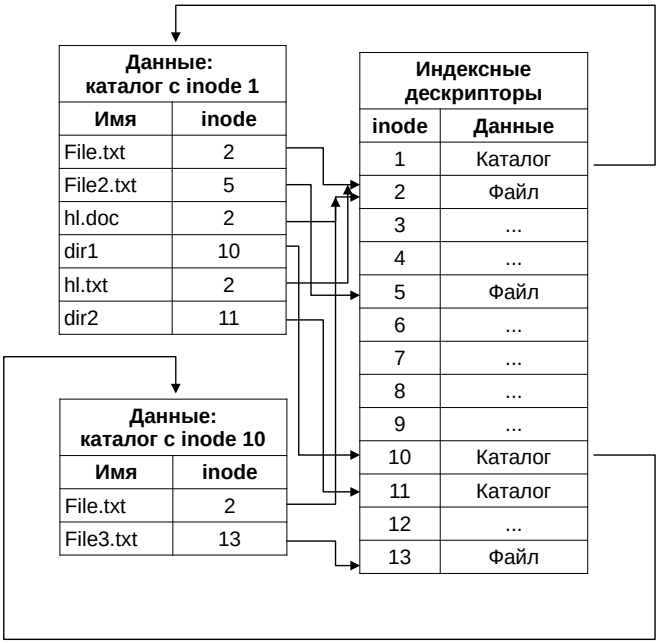
Эти системные вызовы, как для большинства вызовов xv6 (и для большинства системных вызовов — но не их оберточных функций — в Linux), при успехе возвращают нулевое значение (или строго положительное), при ошибке — отрицательное.

```
// Создает новый (пустой) каталог
int mkdir(const char *name);
// Изменяет текущий рабочий каталог процесса
int chdir(const char *name);
// Удаляет файл или (пустой) каталог
int unlink(const char *name);
// Создает жесткую ссылку с именем dst на файл src
int link(const char *src, const char *dst);
```

В xv6, как и в реальных UNIX-подобных системах, всякий файл (в том числе каталог) в ФС накопителя представлен в виде ***индексного дескриптора***

(inode) и записи о нем в каталоге. С индексным дескриптором ассоциированы данные и метаданные файла (к метаданным относится, например, тип файла — регулярный (обычный), каталог, устройство и другие, а также не реализованные в xv6 атрибуты, метки времени, права доступа и др.). Запись в каталоге представляет собой имя файла и номер индексного дескриптора. Таким образом, *всякая запись о файле в каталоге представляет собой (**жесткую**) **ссылку (hard link)** на индексный дескриптор*. Поэтому системный вызов удаления файла называется unlink.

Создание жесткой ссылки на файл представляет собой создание ссылки (с другим именем в том же каталоге или с тем же или другим именем в другом каталоге) на тот же самый номер индексного дескриптора, что и у исходного файла. Таким образом, после этого два «разных файла» оказываются ссылками на одни и те же данные и метаданные, что для упрощения называется «жесткой ссылкой одного файла на другой», хотя формально они равноправны. Рассмотрим пример.



На схеме имеются 4 жестких ссылки на inode с номером 2. Жесткие ссылки на каталоги не допускаются (как и в реальных ОС и файловых системах, однако элементы пути "." и ".." могут реализовываться как жесткие ссылки на себя и на родительский каталог соответственно).

Системный вызов

```
int fstat(int fd, struct stat *st);
```

возвращает информацию о метаданных открытого файла (функция stat работает с помощью open и вызова fstat).

Системный вызов

```
int mknod(const char *name, short minor, short major);
```

Создает файл-устройство. В xv6 в соответствии с принципом UNIX «все есть файл» предполагается, что устройства (аппаратные и виртуальные) представляются в виде особых файлов-устройств. При этом всякое устройство имеет ***старший*** (major) и ***младший*** (minor) номер. Старший номер идентифицирует тип устройства (например, консоль, жесткий диск и т. д.), а значит, и связанный *драйвер* этого устройства. Младший номер идентифицирует конкретный экземпляр устройства данного типа, т. е. конкретное устройство. Данный системный вызов создает файл, однозначно связанный с устройством по его старшему и младшему номеру. Такой файл можно открыть стандартными системными вызовами, а операции ввода/вывода будут перенаправлены в драйвер устройства. В xv6 таким образом реализовано единственное устройство — консоль. В реальных ОС вызов mknod — создания индексного дескриптора и записи в каталоге о нем — является универсальным средством для создания файлов различных типов (регулярных файлов, устройств, именованных каналов, сокетов и других).

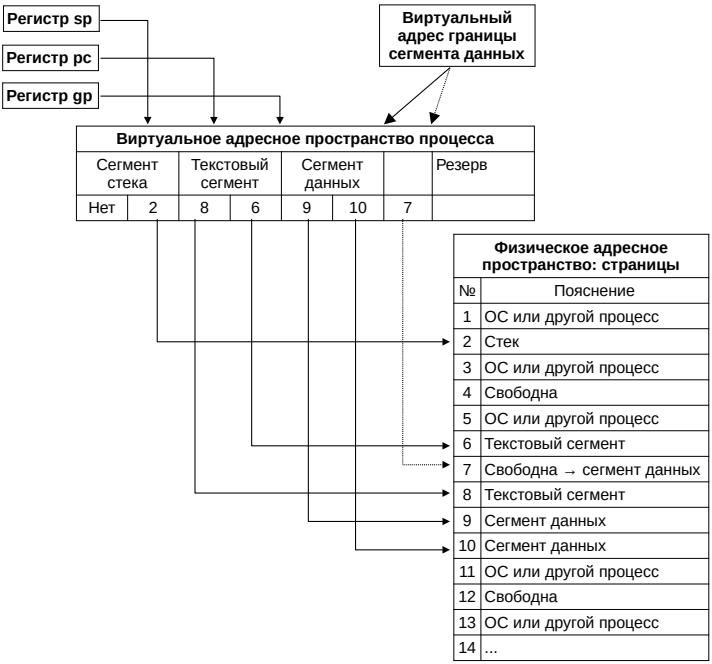
Системные вызовы xv6: другие операции

Системный вызов

```
char* sbrk(int size);
```

изменяет размер сегмента данных на указанную величину. Увеличение размера соответствует выделению дополнительной памяти. Также допускают уменьшение размера сегмента данных с целью освобождения памяти. Фактически системный вызов сдвигает позицию верхней границы выделенного процессу сегмента данных, возвращает указатель на *старую* позицию. Это возможно при использовании страничной адресации памяти без риска фрагментации — при необходимости ядро выделяет новые страницы памяти процессу или освобождает более неиспользуемые страницы с сохранением непрерывности виртуальных адресов границ сегмента данных. При увеличении размера сегмента данных системный вызов вернет указатель на *выделенную память*. В xv6, как и в Linux, нет системных вызовов malloc и free (и их

аналогов) — выделение и освобождение памяти из кучи производится путем сдвига верхней границы сегмента данных, а работа данных функций осуществляется исключительно в пользовательском пространстве. Схематически это можно представить следующим образом.



На схеме показано выделение дополнительной страницы к сегменту данных и изменение значения указателя на его верхнюю границу, хранимую ОС. Отсутствие физической страницы для нижней части стека иллюстрирует здесь один из способов ОС контролировать переполнение стека (попытка обратиться по этому виртуальному адресу приведет к генерации процессором исключения), применяемый в xv6, но не обязательно используемый в реальных системах. В Linux выделение памяти возможно не только с помощью сегмента данных, но и с помощью отображения общего файла или анонимного отображения в память (вызов mmap).

Системный вызов

```
int uptime(void);
```

возвращает время работы ОС от момента запуска. Время вычисляется в «***тиках***» (***tick***) — минимальных квантах времени, через которые срабатывает прерывание системного таймера. В xv6 настраивается на величину в 1000000 циклов процессора. Реальная скорость процессора — количество циклов в се-

кунду — зависит от QEMU и скорости процессора хоста, но один «тик» оценивается в 1/10 секунды.

Системный вызов

```
int sleep(int ticks);
```

приостанавливает исполнение процесса на указанное количество тиков — ожидание без занятия процессорного времени.

Оболочка и пользовательский интерфейс

Программа init, первый процесс, исполняющийся в пользовательском пространстве и запускающий оболочку, представлен в файле `init.c`, а код его запуска — в `initcode.S`.

В xv6 реализована простая оболочка sh, поддерживающая запуск внешних программ (команд) и команду смену каталога (cd). Автодополнение, история команд, переменные окружения, проверка условий и т. п. не поддерживаются.

Из утилит командной строки представлены простые реализации следующих программ:

- echo — печать сообщения;
- cat — вывод файла (одного) в поток вывода;
- wc — подсчет количества байтов, символов и строк в файле;
- grep — поиск по регулярному выражению;
- ls — вывод содержимого каталога;
- kill — отправка сигнала завершения процессу по его идентификатору (pid);
- mkdir — создание подкаталога
- ln — создание жесткой ссылки;
- rm — удаление файла или (пустого) каталога;

Помимо данных POSIX-подобных утилит реализован ряд тестовых приложений:

- usertests — тестирование системных вызовов xv6;
- stressfs — стресс-тестирование файлового ввода/вывода;
- forktest — тест создания и завершения подпроцессов;
- zombie — целенаправленное создание «зомби»-процесса (возможность проверки его «сбора» со стороны init);

- grind — запуск случайных системных вызовов в параллельном режиме.

Большинство утилит не поддерживают опции или поддерживают минимум аргументов командной строки. В то же время их код является достаточно коротким, простым и снабженным достаточным количеством комментариев, поэтому изучение их возможностей и фактической работы предлагается осуществлять путем непосредственного чтения исходного кода.

§3.2. Пространство ядра xv6

Об API пространства ядра

В xv6, как и в Linux, ядро является *монолитным*, причем нет поддержки загружаемых модулей ядра. Это значит, в частности, что из любой функции ядра можно вызвать любую другую. В языке Си, если в единице трансляции функция не помечена как static, то она будет доступна для вызова из другой единицы трансляции, но для корректности генерации кода вызова должен быть доступен ее прототип. *При этом неважно, представлен ли прототип в отдельном* `.h`*-файле или добавлен непосредственно в* `.c`*-файл*. Заголовочные файла — средство удобства и защиты от некоторых ошибок программирования, но они не влияют на работу компоновщика.

Иными словами, изменение функционала монолитного ядра скорее представляет собой модификацию программы, чем написание плагина к этой программе, работающего через фиксированный интерфейс взаимодействия с ней. Монолитные ядра могут не иметь какого-то жесткого API пространства ядра в виде некого «набора функций, которые можно использовать при написании компонентов/модулей ядра, изменения функционала ядра и т. п.». Например, структура этого API, набор многих функций и типов данных, используемых в ядре Linux, сильно менялись и меняются от версии к версии, что требует постоянной актуализации всех модулей.

В xv6 ряд функциональных возможностей ядра вынесен в заголовочные файлы ядра. Среди них имеются следующие:

- `types.h` — определяет типы данных, используемые в xv6;
- `fcntl.h` — определяет флаги открытия файлов с помощью системного вызова open;
- `stat.h` — определяет типы файлов и структуру stat для системного вызова fstat;

- `syscall.h` — определяет символические константы номеров системных вызовов.

Эти заголовочные файлы *подключаются в том числе из программ пользовательского пространства.* Такой подход удобен, т. к. ядро и пользовательское пространство может использовать ряд общих констант и типов данных (хотя и не может использовать общие функции), но не характерен для реальных систем (в т. ч. на базе Linux), где подобные определения типов данных и констант ядра дублируются в заголовочных файлах пользовательского пространства (для библиотеки glibc).

Заголовочный файл `param.h` содержит определение констант для основных параметров системы, таких как лимиты на число запущенных процессов, число процессоров, открытых файлов, устройств и т. д. В целях упрощения под соответствующие структуры ядра — таблица процессов, таблица открытых файлов и т. п. — отводятся статические массивы соответствующей длины, поэтому увеличение этих лимитов приведет к увеличению расхода ресурсов.

Заголовочный файл `defs.h` содержит прототипы функций ядра. Ядро (как xv6, так и реальных систем, например, Linux) не компонуется с библиотекой Си пользовательского пространства (ulib для xv6 и glibc для ОС на базе Linux), в т. ч. потому, что *из ядра нельзя делать системные вызовы* (зато можно вызывать функции ядра напрямую). Поэтому стандартные функции языка Си при программировании пространства ядра недоступны, вместо них доступны только реализованные в ядре. Из таких в `defs.h` определены функции для работы со строками

```
int   memcmp(const void*, const void*, uint);
void *memmove(void*, const void*, uint);
void *memset(void*, int, uint);
char *safestrcpy(char*, const char*, int);
int   strlen(const char*);
int   strncmp(const char*, const char*, uint);
char *strncpy(char*, const char*, int);
```

и функция форматированного вывода на консоль

```
void  printf(char*, ...);
```

смысл которых представляется естественным из заголовка.

Использование функции printf в пространстве ядра нехарактерно для реальных систем в т. ч. потому, что не всякая система всегда работает в консольном режиме. Поэтому в них реализуется более совершенная система протоколирования (логирования) диагностических системных сообщений.

Однако такой подход удобен при изучении xv6. Смысл других функций, объявленных в `defs.h`, завязан на специфические компоненты ядра, поэтому их обзор будет изложен в соответствующих разделах.

В заголовочном файле `riscv.h` содержатся определение архитектурно-специфичных (RISC-V) функций, написанных на языке Си, но с использованием возможности GCC делать ассемблерные вставки — поддержу работы с защищенными регистрами, виртуальной памяти и т. п. Следует отметить, что ядро xv6, в отличие от ядра Linux, не обеспечивает полную структурную изоляцию стандартного кода языка Си и архитектурно-зависимого кода. Последняя особенность делает ядро Linux сравнительно легко портируемым на различные архитектуры процессора, в то время как ОС xv6 более сильно привязана к архитектуре RISC-V.

В других заголовочных файлах ядра объявлены функции и определены структуры данных, также специфичные для отдельных компонентов — файловой системы, драйвера консоли (устройства ввода/вывода), механизмов синхронизации, виртуальной памяти, планировщика процессов и т. д. Эти компоненты описаны в следующих пунктах параграфа.

Запуск системы

При запуске системы управление передается ассемблерной функции _entry (точнее по данной ассемблерной метке в соответствии с настройкой компоновщика), определенной в файле `entry.S`. Это достигается путем помещения кода этой функции средствами настройки компоновщика по адресу передачи управления загрузчиком операционной системы виртуальной машины QEMU.

На многопроцессорной (многоядерной) машине *каждый* процессор (ядро) запускает функцию _entry, и они исполняют ее код параллельно. При этом нужно провести настройку регистров контроля и статуса, в т. ч. организовать трансляцию адреса (виртуальную память), после чего передать управление пользовательским процессам (перейти в пользовательский режим) на каждом процессоре (ядре) независимо.

Функция _entry инициализирует стек ядра и передает управление функции start (файл `start.c`), которая настраивает процессор, в т. ч. таблицы страниц и обработку прерываний, делегируя их в режим супервизора и выставляя адрес обработчика в соответствующий регистр. Эти две функции исполняются в машинном режиме. Функция start передает управление функции main (файл `main.c`) с переводом в режим супервизора (с помощью инструк-

ции mret). Весь дальнейший код загрузки ядра исполняется в режиме супервизора.

Функция main вызывает ряд функций инициализации структур данных ядра ОС. Так как main все еще выполняется на каждом процессоре (ядре) параллельно, сценарий инициализации проводится только на процессоре с номером 0. Этот же сценарий инициализации создает первый пользовательский процесс с помощью функции userinit. Таким образом, *для того, чтобы, например, добавить новые структуры данных — общие массивы, таблицы и т. п. — в ядро, их инициализацию необходимо включить в функцию* main.

Функция userinit *создает пользовательский процесс исполнения файла, собираемого из* user/initcode.S, *который в свою очередь запускает программу, собираемую из* user/init.c, *которая запускает оболочку* sh.

Далее на каждом ядре запускается функция работы планировщика процессов scheduler. При отсутствии готовых к исполнению процессов планировщик переходит в режим ожидания, т. е. ядро простаивает. Все остальные пользовательские процессы запускаются, например, средствами оболочки и другими программами, с помощью системного вызова fork. Они могут исполняться на любом из процессоров, т. е. любой из параллельно работающих планировщиков может выбрать любой ожидающий исполнения процесс.

Переход в пользовательский режим (в т. ч. начало собственно исполнения первого пользовательского процесса — программы init), производится по выходу из обработчика системного вызова или обработчика прерываний, в т. ч. из планировщика.

Обработка ловушек

При срабатывании ловушки (исключения, прерывания, системного вызова) в пользовательском режиме, процессор RISC-V не производит переключение таблицы страниц на таблицу страниц ядра, т. е. работа продолжается в виртуальном адресном пространстве процесса. Для того, чтобы произвести переключение в пространство ядра используются две функции, определенные в файле trampoline.S:

- функция входа в обработчик — userver — сохраняет значения регистров общего назначения в стеке процесса, устанавливает таблицу страниц ядра (значение регистра satp) и передает управление функции usertrap, которая уже исполняется в пространстве ядра;
- функция userret, восстанавливает адрес таблицы страниц процесса в регистре satp, а также регистров общего назначения процесса и вы-

зывает инструкцию sret для возврата в пользовательский режим и передачи управления по нужному адресу.

Сохранение значений регистров процесса производится в специальной странице адресного пространства процесса, называемой TRAPFRAME. Эта страница доступна напрямую из адресного пространства процесса для корректной работы функций uservec и userret, а также из пространства ядра посредством указателя на соответствующую структуру, который сохраняется в специальном поле в записи о процессе в таблице процессов.

Ловушки и прерывания, срабатывающие из пространства ядра, передают управление функции kernelver, сохраняющую регистры в стеке ядра и взывающую функцию kerneltrap, после чего восстанавливающую регистры и вызывающую инструкцию sret. Исключением является обработчик прерывания таймера, выполняющийся в машинном режиме, обрабатываемый функцией timervec. Обе функции определены в файле `kernelvec.S`.

Функции на языке Си, выполняющие непосредственно работу по обработке ловушек, возникших при работе в пользовательском режиме — usertrap — и при работе в режиме супервизора — kerneltrap, определены в файле trap.c. Функции выясняют причину ловушки (регистр scause) и вызывают специфический для каждой причины обработчик. Если это был системный вызов из пользовательского пространства, вызывается функция syscall из файла syscall.c. При возникновении исключения пользовательский процесс принудительно завершается с диагностическим сообщением (дампом регистра sepc, показывающего адрес инструкции, приведшей к ошибке, и регистра scause, содержащего причину ошибки). При возникновении исключения или ошибки в ядре, работа ОС завершается с аналогичным фатальным сообщением об ошибке («паника ядра» — ***kernel panic***). При срабатывании аппаратного прерывания вызывается функция devintr, идентифицирующая устройство, вызвавшее прерывание, по номеру IRQ, и вызывающая соответствующий обработчик.

Обработчик прерывания по таймеру, исполняемый в машинном режиме, планирует следующее прерывание по таймеру и инициирует программное прерывание в режим супервизора. Данное программное прерывание также будет обрабатываться как ловушка и идентифицироваться функцией devintr. Непосредственной обработкой прерывания таймера в этом режиме занимается функция clockintr. Она увеличивает на единицу глобальный целочисленный счетчик тиков ticks. Этот счетчик — общая переменная, поэтому доступ к нему охраняется соответствующей спин-блокировкой tickslock. Кроме того,

обработчик «пробуждает» процессы, спящие в системном вызове sleep, — они продолжат работу, если их тайм-аут приостановки истек.

Код функций-оберток системных вызовов, генерируемый в пользовательском пространстве с помощью user/user.pl в user/user.S, следующий (рассмотрим его на примере функции-обертки open):

```
.global open
open:
 li a7, SYS_open
 ecall
 ret
```

Таким образом, номер системного вызова (здесь — константа из `kernel/syscall.h`) записывается в регистре a7. Все параметры системного вызова передаются через регистры a0 — a6, при этом код передачи генерируется автоматически компилятором языка Си при генерации кода вызова функции-обертки, прототип которой определен в `user/user.h`.

Функция syscall определяет номер системного вызова исходя из *сохраненного в* trapframe *значения регистра* a7. Данный номер используется как индекс в таблице (массиве) системных вызовов syscalls, определенной также в `syscall.c`. Элементами таблицы являются указатели на функции ядра, непосредственно ответственные за обработку системного вызова с данным номером. В данном случае это функция sys_open. Функции обработки системных вызовов в пространстве ядра не имеют параметров и возвращают uint64. Это значение записывается функцией syscall *в качестве сохраненного в* trapframe *значения регистра* a0. После восстановлении значений регистров функцией userret, это значение и будет использовано как возвращаемое значение функции-обертки системного вызова, в соответствии с соглашением о вызовах, используемым компилятором языка Си.

Для получения аргументов системного вызова функции ядра должны извлечь их из сохраненных значений регистров. Это действие может быть выполнено, например, с помощью функций

```
void argint(int num, int *var);
void argaddr(int num, uint64 *var);
```

извлекающих, соответственно, из аргумента номер `num` целое число или адрес в переменную `var`. Номер аргумента соответствует номеру аргументируемого регистра a0 — a8 и должен отслеживаться программистом самостоятельно. *При получении адресного аргумента возвращаемый указатель будет содержать значение виртуального адреса в пользовательском*

пространстве, т. е. не может быть использован напрямую (разыменован) в пространстве ядра, использующим отдельную таблицу страниц. Поэтому используется тип не `void*`, а однозначно приводимый к нему `uint64`. Для того, чтобы получить данные из пользовательского буфера, следует использовать функции ***копирования из пользовательского пространства в пространство ядра*** (чтобы получить, например, входную строку или массив данных) *и* ***обратно*** (если необходимо произвести запись данных в пользовательский буфер, например передать прочитанные системным вызовом read данные). Эти функции относятся к средствам работы с виртуальной памятью и имеют прототипы.

```
// Копирование из пользовательского пространства
// (входные данные системного вызова)
int copyin(pagetable_t pagetable, char *dst,
           uint64 srcva, uint64 len);
// Копирование в пользовательское пространство
// (выходные данные системного вызова)
int copyout(pagetable_t pagetable, uint64 dstva,
            char *src, uint64 len);
```

Здесь `len` — длина копируемых данных (в байтах), `dst` — буфер назначения данных (в пространстве ядра), `srcva` — виртуальный адрес источника (полученный с помощью `argaddr`), `src` — буфер источника данных (в пространстве ядра), `dstva` — виртуальный адрес назначения (полученный с помощью `argaddr`), а `pagetable` — структура диспетчера виртуальной памяти, отвечающая за таблицу страниц процесса. Также можно использовать функцию

```
int copyinstr (pagetable_t pagetable, char *dst,
               uint64 srcva, uint64 max);
```

прочитывающую из пользовательского пространства 0-терминированную строку длины не более чем max символов (включая завершающий `'\0'`), и

```
int argstr(int n, char *buf, int max);
```

напрямую получающую строковый аргумент системного вызова посредством argaddr и copyinstr.

В качестве таблицы страниц текущего (т. е. выполнившего системный вызов) процесса можно использовать `myproc()->pagetable` (функция ядра myproc возвращает указатель на структуру данных, отвечающую за текущий процесс).

Функции прямого получения аргументов из регистра сработают всегда, хотя и могут дать непредсказуемые данные при некорректном коде системного вызова в пользовательском пространстве — они не могут провести проверку наличия и корректности значений аргументов. В то же время функции копирования (в т. ч. argstr) осуществляют проверку корректности переданного из пользовательского пространства адреса: буфер требуемой длины, расположенный по данному адресу, действительно должен лежать в виртуальном адресном пространстве процесса, сделавшего системный вызов. Если эту проверку не выполнить, процесс может злоупотреблять передачей некорректного адреса и попытаться нарушить изоляцию процессов и ядра. Поэтому эти функции логические: возвращают 1 в случае успешного копирования и 0 в противном случае. *Проверка возвращаемого этими функциями значения обязательна. Ошибка их работы должна интерпретироваться как ошибка при осуществлении системного вызова с возвратом последним соответствующего значения (обычно в* xv6 *системные вызовы при ошибке возвращают значение* `-1`). В ОС на базе ядра Linux этот случай соответствует ошибке EFAULT.

Механизмы синхронизации

Поскольку ядро xv6 по умолчанию не только допускает переключение контекста в момент обработки системного вызова, но работает в режиме одновременной мультипоточности, требуется ***синхронизация потоков ядра*** при работе с общими данными ядра и другими общими ресурсами (устройством ввода/вывода, элементами файловой системы и т. д.). Механизмы синхронизации в xv6 представлены ***спин-блокировками*** (структура spinlock, определена в `spinlock.h`) и ***спящими блокировками*** (структура sleeplock, определена в `sleeplock.h`).

- ***Спин-блокировки*** *представляют собой блокировки с активным ожиданием*: они несут меньше накладных расходов на реализацию, годятся для использования в обработчиках прерываний, но тратят процессорное время в процессе ожидания захвата блокировки, поэтому их применение не рекомендуются в тех местах, где ожидание может быть длительным. Реализованы посредством атомарного расширения архитектуры RISC-V.
- ***Спящие блокировки*** *представляют собой блокировки с приостановкой (переводом в спящее состояние) и последующей активацией (пробуждением) процесса, ожидающего захвата блокировки*. Допускают переключение контекста и длительное ожидание, поэтому не го-

дятся для применения в обработчиках прерываний, но предпочтительнее спин-блокировок в других местах, так как позволяют не тратить процессорное время в ожидании захвата блокировки. Реализованы посредством спин-блокировок во избежание конкурентного доступа к полям самой структуры sleeplock и средств планировщика процессов по их приостановке и активации.

Традиционно, xv6 допускает вложенные захваты различных спин-блокировок и спящих блокировок, а также захваты спин-блокировок при захваченных спящих блокировках, но не наоборот. *При вложенных захватах блокировок следует заботиться о недопущении* ***взаимоблокировок*** *— состояний, при которых два и более процесса находятся в бесконечном ожидании друг друга*. Например, если процесс A удерживает блокировку 1 и пытается захватить блокировку 2, в то время как процесс B удерживает блокировку 2 и пытается захватить блокировку 1, они окажутся в состоянии взаимоблокировки.

Функции для работы со спящими блокировками (прототипы объявлены в `defs.h`, функции определены в `spinlock.c`) следующие.

```
// Инициализирует структуру блокировки
void initlock(struct spinlock *lock, char *name);
// Захват блокировки
void acquire(struct spinlock *lock);
// Проверка, удерживается ли блокировка
// (логическая функция)
int  holding(struct spinlock *lock);
// Освобождение блокировки
void release(struct spinlock *lock);
```

Сама структура блокировки должна быть объявлена в глобальной области видимости или при создании экземпляра общей структуры в памяти (т. е. быть общей, как и сам «защищаемый» ей ресурс), а ее инициализация проведена однократно, соответственно, например, при запуске системы (вызвана из функции main) или при создании «защищаемой» структуры. Параметр `name`, задаваемый при инициализации, используется для придания блокировке текстового названия, которое можно использовать в диагностических целях. Таким образом, общая схема применения спин-блокировки следующая:

```
/* Специфический для структуры .c-файл */

// Общая глобальная структура
struct {
```

```
    struct spinlock lock;

    // Поля самой структуры
} some_common_struct;


// Функция инициализации структуры
void init_some_common_struct ()
{
    initlock(&some_common_struct.lock, "scomstr");

    // инициализация полей some_common_struct
}

// В функции работе со структурой
   // ...
   acquire(&some_common_struct.lock);
   // Выполнение работы с полями some_common_struct
   release(&some_common_struct.lock);
   // ...

/* В файле main.c */
// Внутри функции main добавляется строка
  if(cpuid() == 0){
    //
    init_some_common_struct ()
    // ...
  }
```

Если сама общая структура не единственная, а используется какой-то динамический контейнер структур (массив, список — например, создается под каждый процесс, открытый файл и т. п.), то инициализировать надо и сам контейнер, и каждую структуру, добавляемую в него.

Функции для работы со спящими блокировками (прототипы объявлены в defs.h, функции определены в sleeplock.c) аналогичны:

```
// Инициализирует структуру блокировки
void initsleeplock(struct sleeplock *lock, char *name);
// Взятие блокировки
void acquiresleep(struct sleeplock *lock);
// Проверка, удерживается ли блокировка
// (логическая функция)
int  holdingsleep(struct sleeplock *lock);
```

```
// Освобождение блокировки
void releasesleep(struct sleeplock *lock);
```

Общая схема применения спящих блокировок аналогична. Заметим, что функции удаления структур блокировок отсутствуют, т. к. структуры не содержат динамических полей и не требуют освобождения ресурсов при прекращении использования.

Многочисленные структуры ядра и алгоритмы, реализованные в xv6, *используют встроенные или глобальные блокировки для разграничения конкурентного доступа. В целях поддержания краткости и ясности в дальнейшем описании кода ядра в данной главе их наличие опускается. При изучении ядра системы рекомендуется читать и анализировать исходный код, в т. ч. обращать внимание на используемые блокировки.*

Управление процессами

Операционная система xv6 поддерживает многозадачность и многопроцессорные компьютеры, но не поддерживает многопоточность внутри одного процесса. Пользовательские процессы (кроме init) запускаются с помощью системного вызова fork, создающего клон текущего процесса. В Linux копирование страниц памяти для создаваемого дочернего процесса производится только при попытке их изменения одним из процессов, т. е. используется технология ***копирования при записи*** (CoW) следующим образом: записи в таблице страниц (PTE) родительского и дочернего процесса указывают на одни и те же физические страницы, но эти страницы помечаются как защищенные от записи. При попытке записи одним из процессов на такую страницу процессор генерирует исключение, которое операционная система обрабатывает путем создания копии физической страницы и изменения таблиц страниц процессов: у одного из процессов в качестве физического адреса указывается адрес копии, у обоих процессов снимается защита страницы от записи. После этого происходит возврат из обработки исключения на ту же точку — инструкция исполняется повторно. В xv6 не реализована технология копирования при записи, поэтому производится реальное копирование всех страниц виртуального адресного пространства процесса.

Системный вызов exec замещает образ процесса другим исполняемым файлом, в т. ч. подстраивает размер сегментов виртуального адресного пространства процесса, подгружает код и данные нового приложения, очищает место под стек и т. д. (это иллюстрирует пользу использования технологии CoW при fork — если дочерний процесс тут же будет замещен с помощью

exec, копирование подавляющего большинства страниц родительского процесса будет напрасным).

Системный вызов exit закрывает все открытые файловые дескрипторы процесса (как и в реальной ОС, данный вызов должен освободить все ассоциированные ресурсы) и передает управление планировщику процессов.

В планировщике процессов для каждого процесса используется структура proc. Для каждого процессора используется структура cpu. Так как в xv6 не используются динамическое выделение памяти в ядре, то таблицы процессов и процессоров реализованы как статические массивы структур proc и cpu соответственно. Размер этих массивов — максимальное число возможных процессов и процессоров — задается параметрами системы NPROC и NCPU соответственно (файл `param.h`).

Каждый процесс может находится в одном из следующих состояний (`enum procstate`):

- UNUSED — элемент таблицы процессов свободен (не используется), этот статус необходим, т. к. таблица процессов реализована посредством статического массива;
- USED — элемент таблицы процессов используется, но процесс еще не создан (при создании процесса система находит первый попавшийся свободный элемент таблицы в состоянии UNUSED и переводит его в USED, после чего начинает работу над созданием процесса);
- SLEEPING — процесс приостановлен с помощью функции sleep — находится в спящем состоянии (заблокирован в ожидании внешнего события, например, освобождения блокировки с приостановкой, завершения дочернего процесса и др.), может быть переведен в состояние RUNNABLE другим процессом с помощью функции wakeup (механизм описан ниже);
- RUNNABLE — процесс готов к исполнению, но ожидает своей очереди;
- RUNNING — процесс исполняется в данный момент;
- ZOMBIE — процесс в состоянии «зомби» (родительский процесс еще не сделал вызов wait).

Схема переключения между процессами в xv6 следующая.

- Прежде чем процесс передаст управление ядру в результате прерывания или системного вызова, все его регистры будут сохранены в структуре trapframe, отображаемой как в ядро, так и в виртуальное

адресное пространство процесса. Структура содержит регистры общего назначения, сохраненный указатель стека ядра и указатель инструкции, указатель на таблицу страниц ядра (регистр satp) и некоторые другие управляющие регистры. Она представлена на языке Си, имена полей которой соответствуют именам регистров RISC-V (файл `proc.h`).

- Далее исполняется код ядра в контексте текущего процесса. При этом используется ***стек ядра процесса***. Если ядро решает, что в данный момент требуется передать управление ***планировщику процессов***, то ядру необходимо переключить ***контекст внутри ядра*** на планировщик — планировщик процесса исполняется в собственном контексте и в собственном стеке ядра. Это выполняется с помощью функции sched — точки входа в планировщик. Сам процесс планировщика исполняется на каждом процессоре в виде функции scheduler. Она ответственна за выбор очередного процесса и переключении контекста на него: переводит текущий процесс в состояние RUNNABLE (или SLEEPING, если переключение было вызвано не вытеснением по таймеру, а приостановкой, например, с помощью спящей блокировки), выбирает очередной процесс, находящийся в состоянии RUNNABLE и переводит его в состояние RUNNING, после чего делает переключение на его контекст (с помощью описанной ниже функции swtch). Таким образом, xv6 реализует простой циклический алгоритм планирования процессов (*round-robin*). Реальные ОС используют более сложные алгоритмы, учитывающие приоритет процессов, интерактивность, использованную долю процессорного времени и др.
- Переключение контекста внутри ядра — на процесс планировщика или на другой процесс — производит функция swtch. Она реализована на Ассемблере (файл `swtch.S`) и доступна из языка Си через прототип. Аргументами функций являются два контекста — структуры context. Данная структура содержит сохраняемые регистры (s0 — s11) и регистры sp (указатель стека ядра) и ra (адрес возврата) — т. е. те регистры, которые должны сохраняться вызываемой функцией. Действительно, в отличие от срабатывания ловушки, при которой требуется сохранить значения всех регистров (в trapframe), переключение контекста внутри ядра осуществляется просто вызовом функции swtch, сохранение временных регистров — сохраняемых вызывающей стороной — обеспечивается компилятором при генера-

ции кода вызова этой функции. Функция swtch сохраняет значения регистров текущего процесса, а затем загружает значения регистров нового процесса и выполняет инструкцию ret. Так как при этом сохраняется и восстанавливается значение регистра ra — адреса передачи управления при выходе из функции — по ее завершении автоматически происходит передача управления новому процессу.

- С этого момента происходит исполнение кода ядра нового процесса. Он мог быть прерван в системном вызове или при обработке прерывания, исполнение продолжается с этой точки. После завершения обработки прерывания или системного вызова будет восстановлено значение регистров пользовательского пространства данного процесса из его trapframe и передано управление процессу с помощью sret.

Структура proc содержит в себе поля, отвечающие за состояние процесса, его идентификатор, ссылку на родительский процесс, ссылку на таблицу страниц процесса (пользовательское пространство), контекст процесса (сохраненный, на который нужно переключаться с помощью swtch для начала исполнения процесса), ссылку на trapfame процесса, таблицу открытых файлов, код возврата процесса (для передачи родительскому процессу от зомби-процесса), текущий каталог процесса, спин-блокировку доступа к полям структуры процесса, имя процесса и другую информацию. Структура cpu содержит ссылку на структуру proc процесса, исполняемого на данном процессоре, а также сохраненный контекст процесса.

Функция ядра mycpu возвращает указатель на структуру cpu текущего процессора, определяя номер процессора с помощью соответствующей машинной инструкции. Так как в любой момент может произойти переключение между процессами и процесс начнет исполняться на другом процессоре, эту функцию можно вызывать только в режиме отключенных прерываний (это делается с помощью функций push_off — выключение, и push_on — включение; только между вызовами этих функций можно быть уверенными, что номер процессора не изменится; по понятным причинам в таком режиме процессор нельзя оставлять надолго).

Функция myproc возвращает ссылку на структуру proc текущего процесса. Она идентифицирует процесс через идентификацию процессора (с корректным временным запретом прерываний) и может быть использована без дополнительных шагов напрямую.

Приостановка (*sleep*) и активация (пробуждение, *wakeup*) процессов производится с помощью функций sleep и wakeup соответственно. Реализация подобных вызовов «напрямую» сопряжена с риском потерянных активаций,

когда из-за переключения контекста в неудачный момент происходит потеря вызова wakeup и засыпание процесса, который не должен спать. *В литературе такое явление разбирается на примере задачи потребителя и производителя, традиционным решением, применяемым во многих системах и средствах параллельного программирования, является использование* ***семафоров****, осуществляющих подсчет количества отложенных активаций.* В xv6, тем не менее, реализованы именно вызовы для «засыпания» и «активации» без риска потери активаций. Делается это по следующей схеме.

- Засыпание возможно на некотором выбранном канале сна и только в тот момент, когда процессом удерживается выбранная спин-блокировка (lk). Например, если засыпание производится на спящей блокировке, то в качестве канала выступает соответствующая структура sleeplock, а в качестве спин-блокировки lk — встроенная спин-блокировка данной спящей блокировки.
- Вызов sleep блокирует соответствующую структуру процесса на встроенной блокировке и освобождает lk. Процесс переводится в состояние SLEEPING, в его структуре proc сохраняется канал сна и вызывается планировщик (функцией sched) — произойдет переключение контекста. В этот момент lk разблокирован, но заблокирована спин-блокировка приостановленного процесса. Это исключает выполнение функцией wakeup перевода процесса в рабочее состояние до того, как процесс будет полностью переведен в состояние приостановки планировщиком. После работы планировщика sleep разблокирует спин-блокировку процесса и снова захватывает lk.
- Вызов wakeup сканирует все процессы, приостановленные на данном канале сна, и переводит их в состояние RUNNABLE и захватывает lk.

Общая схема применения этих вызовов следующая. Процесс A должен захватить спин-блокировку lk, после этого проверить необходимость перехода в спящее состояние. Если это необходимо, сделать вызов sleep на заданном канале сна с заданным lk. Внутри вызова sleep блокировка lk будет разблокирована, планировщик произведет переключение на другой процесс B. Этот процесс выполнит необходимую работу, после которой станет возможно продолжение исполнение процесса A, и вызовет функцию wakeup. Процесс A продолжит работу в функции sleep, которая снова захватит блокировку lk. По возврату из функции sleep процесс A должен освободить блокировку. Для спящих блокировок в качестве блокировки lk выступает встроенная в струк-

туру sleeplock спин-блокировка, а в качестве канала — сама спящая блокировка.

Такой подход обеспечивает защиту от потерянных активаций. Именно с помощью вызовов sleep и wakeup реализованы спящие блокировки в xv6, не характерные, например, для ядра Linux, где используются блокировки, основанные на семафорах. Аналогичный подход — интерфейс условных ожиданий — присутствует в библиотеке многопоточного программирования POSIX Threads.

В xv6 отсутствуют средства доступа к списку запущенных процессов из пользовательского пространства, а, значит, и утилиты типа ps, однако ядро может напечатать список процессов по горячему сочетанию клавиш `^P`.

Управление памятью

В соответствии с особенностью работы QEMU физические адреса оперативной памяти начинаются с `0x80000000`, а размер оперативной памяти для запуска xv6 настраивается в 128 Мб. Соответствующий диапазон адресов жестко прописывается в код ядра (константы `KERNBASE` — начало диапазона физических адресов и `PHYSTOP` — конец). Реальные операционные системы определяют объем доступной оперативной памяти запросом к аппаратному обеспечению. Физические адреса с меньшими номерами используются для доступа к предоставляемым QEMU загрузчику ОС (Boot ROM), контроллеру прерываний и портам устройств ввода/вывода (консоли и дискового накопителя) посредством отображения в память (MMIO). Константы, ответственные за физическую память и представление виртуальной памяти определены в `memlayout.h`.

Функции, ответственные за выделение физической памяти, определены в `kalloc.c`. Данные функции всегда выделяют и освобождают страницы памяти по одной целиком. Список свободных страниц ведется с помощью структуры run, представляющей собой узел односвязного списка. Сама физическая память представлена структурой kmem, содержащей спин-блокировку списка и указатель на голову списка — стека свободных страниц.

Освобождение страницы (функция kfree) заключается в заполнении ее мусорными данными (в случае xv6 используется код 5 для каждого байта, т. е. чередование 0 и 1) и помещении ее адреса в стек свободных страниц. Заполнение страницы необходимо для удаления из нее данных, которые могли остаться от завершенного процесса — если это не сделать, новый процесс может получить к ним доступ и нарушить изоляцию (в реальных системах это может привести, например, к утечке конфиденциальных данных или паролей

других пользователей). Изначальная инициализация структуры kmem (выполняется функцией kinit) состоит в вызове kfree для всех доступных физических страниц (начинающихся за кодом и статическими данными ядра), т. е. формировании стека всех свободных страниц. Выделение страницы памяти (функция kalloc) состоит в изъятии ее адреса из вершины стека свободных страниц.

Система xv6 использует трехуровневую таблицу страниц SV39. В файле `riscv.h` определен ряд констант, отвечающих за таблицы страниц и виртуальные адреса: `PGSIZE` (размер страницы в байтах, 4096), `MAXVA` (максимально допустимый номер виртуального адреса) и другие. Там же определен тип данных для представления записи в таблице страниц (PTE) — pte_t, представляющий собой 64-битное беззнаковое целое uint64 (в соответствие с архитектурой SV39) и тип данных для представления таблицы страниц — pagetable_t, определенный как указатель на uint64, т. е. моделирующей массив из 512 pte_t размером в одну страницу в 4096 байтов. Таким образом, pagetable_t представляет собой один уровень таблицы страниц.

Для получения информации о записи в таблице страниц — извлечения нужных битов из pte_t — определен ряд макросов и констант, таких как, `PTE_V`, `PTE_R`, `PTE_W`, `PTE_X`, `PTE_U` (биты флагов записи в таблице страниц V — действительная запись, R — право на чтение, W — право на запись, X — право на исполнение, U — право на доступ из пользовательского режима соответственно), `PA2PTE`, `PTE2PA` (макросы для конвертации физического адреса PA в PTE и обратно) и другие.

Функции для работы с виртуальными адресами определены в `vm.c`. Операционная система xv6 создает отдельное виртуальное адресное пространство для каждого процесса и одно виртуальное адресное пространство для ядра. Виртуальное адресное пространство ядра создается функцией kvminit, запускающейся однократно при запуске системы, при этом адрес созданной таблицы страниц ядра верхнего уровня записывается в глобальную переменную ядра kernel_pagetable).

Процедура создания виртуального адресного пространства достаточно прямолинейна. Создание нового виртуального адресного пространства (функция uvmcreate) заключается в выделении единственной страницы для хранения таблицы страниц верхнего уровня (с помощью kalloc) и заполнении ее нулями (заведомо недействительные записи PTE, т. к. их бит V равен 0). Виртуальное адресное пространство процесса представляется именно типом pagetable_t — указателем (массивом) на таблицу страниц верхнего уровня (L2). Поиск физических страниц и записей о них в таблицах страниц, соответствующих заданному виртуальному адресу, а также выделение и создание

их при отсутствии (в т. ч. для записи таблиц страниц нижних уровней L1 и L0) осуществляется с помощью функции walk. Эта функция получает на вход таблицу страниц верхнего уровня и виртуальный адрес, на выходе возвращает соответствующую PTE или 0, если адрес недействителен. Функция работает циклическим проходом по таблице страниц от верхнего до нижнего уровня. Если включен (задан как не ноль) параметр-флаг alloc этой функции, то при отсутствии страницы, соответствующей виртуальному адресу или хранящей таблицу страниц уровня L1 или L0, происходит ее создание с помощью kalloc. Таким образом, как конвертация виртуального адреса в физический, так и создание страниц для процесса (функция mappages) с соответствующим дополнением и модификацией таблицы страниц, производится с помощью функции walk.

Создание виртуального адресного пространства «с нуля» производится только для первого процесса — init. Во всех остальных случаях используется fork, вызывающий копирование виртуального адресного пространства с помощью функции uvmcopy. В дальнейшем системный вызов brk может использовать uvmalloc и uvmdealloc для, соответственно, расширения и сужения адресного пространства процесса. Системный вызов exec аналогично использует эти функции для подстройки размера виртуального адресного пространства под размер сегмента данных и кода нового исполняемого файла. При завершении процесса адресное пространство очищается функцией uvmfree. Функции, освобождающие виртуальную память, проходят рекурсивно по таблице страниц, вызывая kfree для каждой освобождаемой страницы.

Функция walkaddr посредством функции walk проводит преобразование виртуального адреса в физический (т. е. фактически программно выполняет трансляцию адреса), проверяя при этом действительность записи (флаг V) и право пользовательского доступа (флаг U) к соответствующей странице. Напомним, что процессор при исполнении процессов осуществляет такое преобразование автоматически без вмешательства и информирования ОС в соответствии с настроенной таблицей страниц. Однако такое программное преобразование нужно ОС для работы функций copyin, copyout и copyinstr. *Заметим, что блок памяти в виртуальном адресном пространстве может пересекать границы физических страниц, поэтому этим функциям недостаточно преобразовать только начало заданного блока — необходимо отслеживать конец каждой физической страницы и проводить преобразование при каждом переходе на новую физическую страницу.*

Можно заметить, что при работе вышеописанных функций используются строковые функции Си пространства ядра, такие как memset, memmove и

др. Реализация этих функций произведена через обычную работу с указателями, т. е. они *работают с виртуальными адресами пространства ядра*. При этом в функциях работы с виртуальной памятью они получают в качестве аргумента *физические адреса*. Это не приводит к путанице потому, что *таблица страниц ядра настраивается таким образом, чтобы физические адреса транслировались в себя*. В реальных операционных системах могут использоваться архитектурно-специфические инструкции копирования из одного физического адреса в другой и т. п.

Виртуальная файловая система и файловый ввод/вывод

В xv6, в соответствии с принципом UNIX, известным как «*все есть файл*», посредством интерфейса файлового ввода/вывода доступны не только собственно файлы на накопителях информации, но и другие объекты: устройства и каналы. В реальных ОС, например, на базе Linux, в виде файлов представляются еще и ***сетевые сокеты*** (служащие для обмена данными между процессами ***клиента*** и ***сервера*** по сети), общая память и другие объекты. Таким образом, в таких системах файл является базовой абстракцией (парадигмой) представления, хранения, передачи данных и операций ввода/вывода.

В xv6 поддерживается только собственная файловая система накопителя в единственном экземпляре — корневая файловая система виртуального диска, не разбиваемого на разделы. Монтирование файловых систем не поддерживается. Также в отличие от реальных систем, например, на базе Linux, в xv6

- не поддерживаются именованные каналы — файлы особого типа, которые позволили бы осуществлять межпроцессное взаимодействие несвязанным иерархией наследования процессам;
- поддерживаются только ***символьные устройства ввода/вывода***, но не ***блочные устройства***, *хотя поддерживается блочный ввод/вывод на единственный виртуальный накопитель, называемый* ***виртуальным диском***;
- не поддерживаются ***символические ссылки (symbolic link)***;
- не поддерживаются ни на уровне ФС раздела, ни на уровне виртуальной файловой системы права доступа к файлам, метки времени обращений к файлам, атрибуты и другие метаданные файла.

Задача файловых систем блочных устройств — размещение файлов по блокам накопителя данных — не является тривиальной. Простое сохранение

данных всех файлов «подряд» и ведение списка точки начала и размера каждого файла возможно только до тех пор, пока не будет производится добавление данных в конец существующих файлов или удаление файлов из середины. В этом случае поддержание тривиальной структуры потребует переноса больших объемов данных с одной части носителя информации на другую. Поэтому обычно ФС размещают файлы фрагментами, распределяя их по блокам. При этом каждый файл хранится в виде целого количества блоков (занимая несколько больше пространства накопителя, чем его реальный размер), а ФС хранит для каждого файла список (порядок, цепочку) номеров блоков, в котором размещены данные файла, а также его фактический размер (данные последнего блока списка, следующие за концом файла, определенным хранимым размером, являются «мусорными»).

В xv6 можно выделить следующие уровни абстракции для доступа к файлам (от самого низкого к самому высокому):

- ***дисковый уровень*** осуществляет операции с дисковыми (физическими) блоками данных (соответствующие функции определены в `virtio_disk.c`);
- ***уровень кеширования*** сохраняет данные блоков в оперативной памяти, позволяя сократить количество операций физического ввода/вывода (функции определены в `bio.c`);
- ***уровень журналирования*** призван обеспечить сохранность файловой системы при сбоях и прерываниях работы ОС, например в следствие фатальной программной ошибки или потери электропитания (определен в `log.c`);
- ***уровень индексных дескрипторов (inode)*** обеспечивает работу непосредственно с данными и метаданными файла на диске (основные функции определены в `file.c`);
- ***уровень каталога (директории)*** обеспечивает представление каталога как особого inode, данные которого представляют собой ссылки на файлы (в т. ч. подкаталоги), т. е. пары вида «имя файла» — «номер inode» (основные функции определены в `fs.c`);
- ***уровень путей и имен файлов*** (может рассматриваться как аналог VFS) обеспечивает доступ к файлам по их уникальным именам в ОС (основные функции определены `fs.c`);
- ***уровень файловых дескрипторов*** обеспечивает доступ как к регулярным файлам посредством системных вызовов (системные вызовы

определены в sysfile.c, а основные операции — в file.c), так и к открытым каналам, и символьным устройствам.

Ввод и вывод с помощью каналов (pipe) является уровнем абстракции следующим ниже уровня файловых дескрипторов и определен в pipe.c. Каждый канал реализован как ограниченный кольцевой буфер данных (в соответствии с классическим решением задачи производителя и потребителя — процессы блокируются, если запись невозможна из-за заполненного буфера, и в ожидании поступления данных при чтении из пустого буфера). Операции с единственным символьным устройством — консолью — в console.c. Таким образом, и обработчик системного вызов sys_open, и обработчик sys_pipe, создают в массиве дескрипторе открытых файлов процесса структуру file (file.h).

В этой структуре указывается тип открытого файла, соответствующего дескриптору — FD_NONE (дескриптор не используется процессом — такой тип необходимо поддерживать из-за того, что в xv6 используются статические массивы), FD_PIPE (канал), FD_INODE (регулярный файл), FD_DEVICE (символьное устройство). Там же сохраняется или ссылка на структуру pipe (если это канал), или структуру inode (если это регулярный файл), или старший (major) номер устройства (если это символьное устройство). В той же структуре записываются флаги, показывающие ли дескриптор для записи и чтения для контроля возможности проведения соответствующей операции. Для inode также хранится точка смещения в файле (off) с которой следует производить следующие операции ввода/вывода (в xv6 не реализованы операции позиционирования, поэтому весь ввод и вывод является последовательным, но для его обеспечения хранение необходимо хранение текущей позиции).

При осуществлении ввода/вывода функции-обработчики sys_read и sys_write делегируют запросы функциям fileread и filewrite соответственно, которые, в зависимости от типа файла, перенаправляют их в piperead/pipewrite, readi/writei и функциям, адрес которых хранится в указателе read/write специального массива devsw — таблице символьных устройств, в качестве индекса в которой используется старший номер устройства. Таким образом, xv6 позволяет регистрировать новые типы устройств ввода/вывода путем разработки соответствующих функций и заполнения указателей данной таблицы.

В xv6 нет специальных функций для чтения содержимого каталога. Вместо этого каталог открывается как регулярный файл, в который сохраняются структуры dirent (fs.h), содержащие имя файла (массив символов фиксиро-

ванной длины) и номер inode (именно путем прочтения таких структур из файла-каталога работает утилита ls).

Для каждого обрабатываемого файла в ядре хранится структура inode (`file.h`), содержащая информацию о номере inode в дисковой файловой системе, размере файла и списке его блоков (для регулярного файла) старшем и младшем номере устройства (для устройств), количестве жестких ссылок на файл и другую информацию. Эта структура является копией в оперативной памяти фактической записи о inode на диске. Дисковая запись inode представлена в виде структуры dinode (`fs.h`), т. е. строго говоря эти структуры — inode в ядре и inode в файловой системе — не являются тождественными. Структура dinode записывается на диск и прочитывается с него напрямую, как двоичные данные, для обеспечения работы уровня индексных дескрипторов дисковой ФС. Структура inode создается и заполняется при открытии файла для обеспечения работы уровня файловых дескрипторов ОС. Именно к ней обращается системный вызов fstat для получения информации о файле. Разрешением полных и относительных путей, рекурсивным проходом от корневого каталога к файлу и чтением dinode с диска занимаются функции namei, namex и другие (`fs.c`).

Дисковая файловая система и журналирование

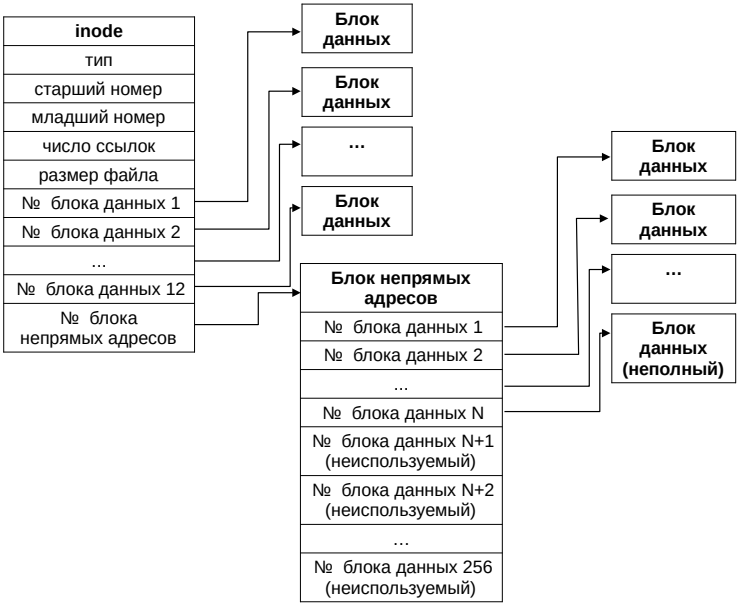
Для работы xv6 создается виртуальный жесткий диск размером в 2000 блоков по 1024 байта каждый (размер задается в `param.h` и используется как ядром, так и утилитой mkfs). Файловая система, создаваемая на нем, имеет следующую структуру:

- нулевой блок содержит загрузчик системы, при старте компьютера он прочитывается и исполняется как машинный код, его задача — найти ядро ОС, загрузить его в память и передать ему управление (при запуске xv6 QEMU автоматически загружает файл с кодом ядра, указываемый в качестве параметра командной строки, по фиксированному адресу, а загрузчик передает по нему управление);
- первый блок, называемый (как и во многих реальных ФС, и в ядре Linux) ***суперблок*** (***superblock***), содержащий метаданные файловой системы — размер файловой системы, число индексных дескрипторов, размер журнала, количество блоков данных и др.;
- начиная с блока 2 следует журнал файловой системы (занимает фиксированное количество блоков), описанный на стр. 147;

- после журнала следуют таблица индексных дескрипторов (inode, их количество фиксировано, т. е. число файлов, которое можно создать в рамках ФС не может превышать этого значения; подобное ограничение характерно и для некоторых реальных ФС, например используемой в Linux ФС ext4) — каждый inode содержит данные, содержащие метаданные файла (тип, количество жестких ссылок, размер, старший и младший номер устройства, адреса блоков данных файла — в реальных системах к ним же относятся права доступа, метки времени и другие атрибуты), знание номера inode позволяет быстро найти и прочитать данные из таблицы;
- после таблицы inode следует битовая карта свободных и занятых блоков данных, позволяющая быстро находить свободные блоки для записи данных файлов;
- по окончании этой таблицы следуют собственно блоки с данными файлов.

Процесс создания файловой системы, включающий в себя запись загрузчика, суперблока и корневого каталога, осуществляется с помощью утилиты mkfs, исполняемой в процессе сборки ОС — эта программа создает файл-образ виртуального диска, файловую систему на нем, и копирует на нее файлы утилит пользовательского пространства ОС xv6. Двоичный файл ядра ОС на виртуальный диск не записывается, он передается виртуальной машине QEMU как отдельный параметр командной строки.

Структура элемента записи inode содержит место для 12 адресов блоков, что позволяет хранить файлы размером до 12 килобайтов, а также одно место для блока ***непрямой адресации (indirect block)*** — это блок, который хранится в области блоков данных файлов, но содержит адреса блоков файла (до 256 блоков, т. к. номер блока 32-разрядный). Таким образом, можно сохранить еще 256 килобайтов данных в файле (т. е. размер файла в xv6 ограничен 278 килобайтами). Это довольно небольшое значение по меркам современного мира, но достаточное для работы xv6. Схема структуры представления inode на диске следующая:



Количество блоков файла может быть меньше максимально возможного, поэтому последние записи таблицы блоков не используются (содержит «мусорные» данные). Размер файла может быть не кратен размеру блока, поэтому последний блок файла используется частично (после конца файла располагаются «мусорные» данные).

Сохранение количества жестких ссылок позволяет определить, используется ли еще данный inode, т. е. нужно ли еще хранить его данные или блоки данных могут считаться свободными: при создании inode (файла) — вызове mknod — это значение устанавливается в 1, при создании жесткой ссылки на тот же inode, т. е. записи в каталоге — вызове link — это число увеличивается на 1, а при удалении файла — вызове unlink — уменьшается на 1. При достижении 0 данный inode считается удаленным и все соответствующие блоки данных помечаются как свободные.

Все метаданные файловой системы (inode и битовые блоки) сохраняются на диск в двоичном виде. При этом в общем случае требуется учитывать порядок байтов, но т. к. реализация xv6 существует только для архитектуры одного типа, конвертации байтов в ней не производится. Однако, при разработке драйвера файловой системы xv6 для других ОС необходимо учесть, что числовые метаданные де-факто сохраняются в ней в обратном порядке байтов (в соответствие с архитектурой RISC-V).

Если в процессе работы ОС возникнет сбой (например, из-за фатальной ошибки в ядре, прерывании электропитания или остановки виртуальной машины) существует риск, что манипуляции с файловой системой будут произведены не полностью, что может привести ее в неконсистентное состояние. Например, в каталоге может быть ссылка на незаполненный inode, блок данных, реально используемый существующим inode, окажется помеченным в битовой карте как свободный, нарушена структура записей содержимого каталога и т. п. В этом случае работа с файловой системой может привести к серьезной потере данных.

Одно из возможных решений проблемы — проводить при запуске ОС проверку целостности файловой системы и исправление ошибок, что делалось на ранних ОС и ФС. Это решение, однако, не является оптимальным, так как исправление ошибок не всегда может быть выполнено, что приведет к потере данных. Причем могут оказаться потеряны не только данные файлов, действия с которыми производились в момент сбоя, но и данные файлов, косвенно затронутых сбоем — например, если их структуры ФС оказались размещены на ставших некорректными блоках.

Более качественную защиту от потери данных предоставляет механизм ***журналирования (journaling) файловой системы*** — до выполнения операций со структурами ФС следует записать предстоящие операции в журнал, после чего выполнить их. Обычно это относится только к операциям со структурами ФС (создание и удаление файлов и каталогов, выделение и освобождение inode, выделение и освобождение блоков данных, модификация inode — метаданных файлов и адресов блоков данных и т. п.). Журналирование операций с данными файлов приведет к дублированию всех данных в журнале и к более чем двукратному снижению скорости записи.

Обычно (в т. ч. в xv6) журналы организованны по принципу ***транзакций*** *— любая операция над структурой файловой системы, состоящая из ряда манипуляций со служебными блоками файловой системы, должна быть либо завершена полностью либо не была начата вообще*. То есть не атомарная — состоящая из модификаций нескольких блоков — операция должна быть превращена в атомарную таким образом, чтобы ни при каких обстоятельствах не было возможности прерывания на середине ее выполнения. Например, создание нового файла требует изменения блока данных каталога путем внесения записи о нем в блок данных каталога (это может потребовать выделения дополнительного блока данных каталога с модификацией inode каталога и битовой карты занятых блоков, если в последнем блоке данных каталога не осталось места), нахождение свободного inode для создаваемого

файла и заполнении данных в нем. Эти операции должны быть или выполнены все полностью, либо не выполнена ни одна из них.

Для осуществления журналирования все операции сначала вносятся в журнал, для этого перед их началом вызывается функция begin_op (`log.c`), отмечающая начало транзакции, выполняются записи в журнал с помощью log_write, по завершении операции вызывается функция end_op, вносящая в журнал запись о завершении транзакции и выполняющая функцию commit, которая выполняет запланированные операции с боками ФС и удаляет запись о транзакции из журнала. Если исполнение операции будет прервано, то при загрузке системы в журнале или будет обнаружена незавершенная транзакция (end_op не был вызван), тогда такая запись удаляется и действия над ФС не производится вовсе, либо будет обнаружена завершенная, но не выполненная транзакция (end_op вызван, но commit не успел завершить свою работу). В этом случае записанные в журнале операции выполняются — это приводит блоки в то состояние, в которое они должны прийти по выполнении транзакции (даже если часть операций уже была выполнена в процессе прерванной работе commit), то есть операция выполняется полностью.

Журналирование в xv6 может показаться избыточным, т. к. не предполагается сохранения на виртуальном диске важных данных, более того, образ диска пересоздается каждый раз при сборке ОС. Однако реализованное средство журналирования хорошо иллюстрирует принципы организации этого механизма в реальных журналируемых файловых системах.

Блочный ввод/вывод и кеширование

Операции ввода/вывода, в т. ч. дискового, являются медленными, требующими обращения к контроллерам устройств и передачи данных физическому носителю, поэтому ОС реализуют механизм кеширования для сокращения числа таких операций. Действительно, если блок данных *уже был прочитан* одним процессом, то пока он храниться в кеше, нет необходимости снова прочитывать его в случае повторного запроса этим же или другим процессом. Если блок был *только что записан* одним процессом, то пока его копия хранится в кеше, нет необходимости обратно прочитывать его при запросе содержимого этим же или другим процессом. Если блок был только что модифицирован одним процессом, то нет необходимости *сражу же* сохранять его на диске, так как он может быть в ближайшее время снова модифицирован этим же или другим процессом.

Известно, что одним из наиболее эффективным алгоритмом кеширования является алгоритм вытеснения блока, который не использовался дольше

всего (LRU, ***Least Recently Used***). Именно такой алгоритм применяется в xv6 (`bio.c`). Функция bread прочитывает блок с указанным номером с диска, после чего блок остается зарезервированным как используемый и его кеш сохраняется в памяти. Функция bwrite изменяет содержимое блока в памяти. Функция brelse освобождает блок, помечает его как неиспользуемый. Для контроля очередности использования блоков xv6 ведет их двусвязный список, помещая в начало используемые блоки. Размер кеша (количество блоков) ограничено.

При нехватке места в кеше для чтения очередного блока удаляется тот блок, который оказался в конце списка. Функция bget прочитывает содержимое блока путем поиска блока с соответствующим номером в списке. Это не самое эффективное решение, более эффективным (стандартным) подходом является ведение в дополнение к связному списку хеш-таблицы, сопоставляющей номеру блока ссылку на элемент списка. Также в реальных системах хорошо зарекомендовали себя двухуровневые LRU кеши — т. е. ведение двух очередей, верхней и нижней. Новые блоки помещаются в голову верхней очереди, хвостовые блоки верхней очереди помещаются в голову нижней, а удаляются хвостовые блоки нижней очереди.

На практике ситуации последовательной работы с одним и тем же дисковым блоком встречаются достаточно часто, поэтому кеширование является очень эффективным механизмом оптимизации. *Заметим, что использование кеша записи (когда применяется отложенная запись на диск) может нести опасность потери данных, если работа операционной системы будет прервана до тех пор, пока данные были реально записаны на диск, в то время как использование кеша чтения не несет дополнительных рисков.* С другой стороны при отсутствии данных в кеше при чтении, требуется загрузить их с диска как можно быстрее, чтобы избежать простоя ожидающего их приложения. В то же время при записи синхронизация кеша с диском может производится «лениво» (быть отложенной), так как в этом случае приложение имеет возможность продолжить работу сразу после того, как выводимые данные поступили в кеш.

Аппаратные прерывания и драйверы физических устройств

Для того, чтобы операционная система могла исполняться на виртуальной машине или реальном оборудовании необходима поддержка соответствующей аппаратуры — видеокарты, клавиатуры, мыши, сетевые карты, принтеры, аудиокарты, накопители информации и др. Программное обеспече-

ние, осуществляющее взаимодействие операционной системы с оборудованием, обычно называется ***драйвером устройства***.

Многообразие оборудования делает задачу создания ОС совместимой с широким спектром компьютеров — хотя бы даже одного класса компьютеров (например, настольных, портативных, мобильных и т. д.), одной архитектуры и одного поколения — достаточно сложной. Именно драйверы различных устройств занимают значительную часть (более 2/3) кода ядра операционной системы Linux. Ядро Linux поддерживает очень широкий спектр компьютерных систем, от встраиваемых до суперкомпьютеров. Однако даже этого недостаточно — значительная часть оборудования (особенно, устанавливаемого на портативные и мобильные компьютеры) не имеет драйвера, включенного в ядро Linux, в силу закрытости спецификации или отсутствия временных ресурсов у сообщества для его разработки.

Под термином «драйвер» может пониматься не только драйвер аппаратного устройства, но и драйвер виртуального устройства (например, драйвер виртуального привода оптических дисков, позволяющий имитировать наличие на компьютере такого привода даже при его физическом отсутствии для тех программ, для которых наличие диска обязательно для работы), драйверы файловых систем, драйверы псевдоустройств (например, драйвер устройства /dev/urandom в Linux, чтение из которого выдает псевдослучайные последовательности байтов) и др.

Большинство операционных систем позволяет подключать драйверы устройств посредством специального интерфейса или в виде загружаемых модулей ядра (Linux). При монолитной архитектуре ядра системы код драйверов исполняется в пространстве ядра, при микроядерной — в пользовательском пространстве, но обработка прерываний все равно осуществляется в пространстве ядра. Использование кода драйверов сторонних производителей в пространстве ядра может служить источником фатальных ошибок и уязвимостей. Ядро xv6, как и Linux, является монолитным.

ОС xv6 не предоставляет какого-либо интерфейса для подключения драйверов устройств, код драйверов интегрирован в код ядра. Реализована работа драйверов двух виртуальных устройств, предоставляемых QEMU:

- UART консоль (Universal Asynchronous Receiver/Transmitter — универсальный асинхронный приемопередатчик) — разновидность серийной консоли, позволяющая вводить и выводить текстовую информацию; со стороны операционной системы хоста ввод/вывод в эту консоль преобразуется в простейший ввод/вывод в текстовый терминал или его эмулятор;

- virtio — стандарт виртуальных устройств ввода/вывода для виртуальных машин (дисковых, сетевых и др.), в xv6 реализован в виде виртуального накопителя данных («виртуального аналога жесткого диска»); со стороны операционной системы хоста представлен в виде файл-образа диска `fs.img` соответствующего размера.

Аппаратным устройствам как правило присваивается уникальный номер аппаратного прерывания (IRQ). По этому номеру ОС может определить, какое именно устройство вызвало прерывание и обработать возникшую операцию ввода/вывода. За присвоение номеров аппаратных прерываний отвечает контроллер прерываний, зависящий от архитектуры процессора и производителя главной платы компьютера. В RISC-V под QEMU в качестве контроллера прерываний используется виртуальный контроллер PLIC (RISC-V Platform Level Interruption Controller — Контроллер Прерываний на Уровне Платформы RISC-V).

Общение с PLIC осуществляется с помощью функций файла `plic.c`. Обработка аппаратных прерываний передается в функцию devintr (`trap.c`). Номер прерывания (IRQ) определяется с помощью функции plic_claim, которая в свою очередь обращается к соответствующему физическому адресу в памяти. Порты и регистры контроллера прерываний, как и контроллеров устройств ввода/вывода, отображаются в память в диапазон физических адресов от `0x0` до `0x80000000` (исключительно — начиная с данного адреса следуют ячейки физической памяти). Адреса, соответствующих контроллерам устройств ячеек, памяти и номера прерываний определены в `memlayout.h`. Функция devintr передает управление uartintr и (`uart.c`) или virtio_dist_intr (`virtio_disk.c`) в соответствие с этим номером. В этих же файлах (`uart.c` и `virtio_disk.c`) реализована остальная логика работы с устройствами UART и VIRTIO_DISK соответственно.

ОС xv6 не поддерживает подключение нескольких устройств UART и VIRTIO_DISK. В реальной системе несколько идентичных устройств получали бы разный идентификатор или даже номер IRQ, но обрабатывались бы одним и тем же драйвером. Единственное поддерживаемое устройство каждого типа имеет жестко кодированный номер 0.

Обработчик прерывания UART прочитывает из устройства очередной символ — именно нажатие клавиши на консоли генерирует прерывание — и передает драйверу консольного ввода/вывода xv6 — функции consoleintr (`console.c`). Эта функция либо помещает введенный символ в глобальный буфер ввода консоли (поле cons.buf), либо немедленно обрабатывает нажатую клавишу, если это горячая клавиша ядра системы (например, ^P — вывод

списка процессов). Таким образом, при нажатии клавиши (наборе на клавиатуре) данные сначала поступают в буфер в памяти ядра в асинхронном режиме (с помощью аппаратных прерываний — без периодического опроса клавиатуры на предмет наличия манипуляции со стороны пользователя), откуда потом прочитываются с помощью функции cosoleread в соответствующий файловый дескриптор при запросе со стороны приложения. Это типичная логика ввода, основанного на аппаратном прерывании.

Вывод информации в консоль производится с помощью функции передачи в функцию драйвера uartputc, которая уже обеспечивает немедленную передачу символа в программный буфер uart_tx_buf и последующей передачей символа на устройство UART. При выводе аппаратные прерывания не используются. При вводе, если программный буфер консоли полон — то есть пользователь нажимает клавиши быстрее, чем система (приложения) успевают их обработать, то очередной символ можно просто проигнорировать. В то же время при заполнении буфера вывода игнорировать очередной символ нельзя — требуется блокировка выводящего процесса в ожидании освобождения буфера.

Операции и ввода, и вывода с диском VIRTIO осуществляются с помощью прерываний. Для этого функция virtio_disk_rw запрашивает ввод или вывод дискового блока и приостанавливает (sleep) текущий процесс. По завершении операции генерируется прерывание, которое пробуждает приостановленный процесс (wakeup). Это обеспечивает освобождение процессора процессом на период ввода/вывода, с возможностью предоставить процессорное время другому процессу. Это приводит к тому, что ввод и вывод в xv6 является блокирующим, т. е. системные вызовы read и write приостанавливают работу программы в ожидании ввода и вывода данных. Аналогично поступает и функция consoleread, если буфер пуст. В реальных ОС, в т. ч. в ядре Linux, существуют неблокирующие варианты системных вызовов ввода/вывода, когда работа программы может быть продолжена в ожидании вводимых данных или завершения вывода.

Глава 4. Лабораторные работы и задачи

В главе описываются получение, компиляция, запуск и отладка ОС xv6, создание программ на Ассемблере RISC-V и языке Си в пользовательском пространстве, добавление системных вызовов и компонентов ядра ОС данной операционной системы, а также приводится ряд лабораторных заданий на

программирование компонентов ядра и пользовательских программ на различные темы (создание системных вызовов, обмен данными между пространством ядра и пользовательским пространством, переключение и синхронизация процессов, управление памятью, файлы и файловые системы), указываются пути их решения и моменты, которые следует учесть при реализации решений.

Дополнительную информацию и задачи можно получить на странице курса MIT 6.1810, посвященном операционным системам. Описание последней на момент реализации на данного пособия курса (осенью 2023 года) доступно по адресу `https://pdos.csail.mit.edu/6.828/` (дата доступа 01.02.2024).

Примеры исходных кодов, приведенных в данном пособии, размещены автором по адресу `https://github.com/lovyagin/xv6-riscv-samples` (дата доступа 01.04.2024) в виде ответвления от основного кода xv6. Исходный код xv6 может развиваться независимо от примеров из данного пособия, поэтому репозиторий и сами примеры в нем могут устаревать. При решении задач рекомендуется использовать актуальную версию исходного кода xv6 от разработчиков, ориентируясь на ответвление, сделанное для данного пособия, только в целях изучения примеров.

§4.1. Подготовка к работе

Установка необходимого ПО

Перед началом работы с операционной системой xv6 необходимо установить пакеты, обеспечивающие кросс-платформенную компиляцию (gcc), отладку (gdb) и запуск виртуальной машины (QEMU) под архитектуру RISC-V и необходимые для разработки утилиты (make, git). Способ установки и конкретные имена пакетов зависят от операционной системы хоста. Также процедура может варьироваться в зависимости от версии ОС хоста и модификации xv6, поэтому рекомендуется следовать актуальным руководствам на странице курса MIT (на момент написания пособия это версия руководства, доступная по адресу `https://pdos.csail.mit.edu/6.828/2023/tools.html`, дата доступа 01.02.2024) и документации к используемой ОС хоста.

На момент написания пособия пользователям дистрибутива Debian (и, вероятно, основанных на них, таких как Ubuntu, и др.) рекомендуется установить пакеты командой

```
$ sudo apt-get install git build-essential
  gdb-multiarch qemu-system-misc gcc-riscv64-linux-gnu
  binutils-riscv64-linux-gnu
```

Для дистрибутива Arch Linux рекомендуется использовать

```
$ sudo pacman -S riscv64-linux-gnu-binutils
  riscv64-linux-gnu-gcc riscv64-linux-gnu-gdb
  qemu-emulators-full
```

Руководство для пользователей ОС Fedora не предоставлено. Для этого дистрибутива установка пакетов для компиляции и запуска xv6 может быть выполнена командой

```
$ sudo dnf install git gdb make gcc qemu-system-riscv
  gcc-riscv64-linux-gnu binutils-riscv64-linux-gnu
```

На момент написания пособия дистрибутив Fedora не содержит отладчика gdb для платформы RISC-V. Можно воспользоваться, например, пакетом riscv64-elf-gdb из дистрибутива Arch Linux, взяв его установочный файл. Установить необходимые для его работы и распаковки зависимости

```
$ sudo dnf install zstd guile30 python3.11-libs
```

и извлечь из установочного файла пакета — как из архива — исполняемый файл riscv64-elf-gdb. Именно этот файл следует запускать вместо gdb для отладки xv6. *Работоспособность способа не гарантируется, читателю следует понять идею и распространить ее на случай отсутствия тех или иных пакетов в используемом дистрибутиве, гибко анализируя зависимости и находя необходимые файлы и пакеты, желательно, в официальных репозиториях используемого или других популярных дистрибутивов ОС на базе Linux; на момент написания пособия в поиске пакетов может помочь сайт pkgs.org (дата доступа 01.02.2024).*

Пользователи ОС Windows могут воспользоваться Windows Subsystem for Linux (WSL) версии 2 или выше, установив последнюю версию Ubuntu в ней. Далее провести установку стандартных пакетов в соответствие с руководством для дистрибутива на базе Debian. ОС xv6 следует скачивать в домашний каталог WSL (`/home/`*username*) средствами командой строки WSL (git), так как для ее сборки и работы требуется поддержка POSIX прав доступа (право на исполнение). Доступ к файлам WSL из `Windows`-приложений может быть осуществлен по пути `\\wsl$\`, что, например, позволяет вести разработку для xv6 привычными графическими средствами. При этом компиляция и запуск xv6 следует производить средствами WSL, а при использовании Windows-приложений для редактирования текста отслеживать, чтобы

файлы исходного кода xv6 сохранялись с переводом строк байтом LF (Unix), а не комбинацией CRLF (DOS/Windows)

Инструкция для пользователей macOS, приведенная на вышеуказанной странице сайта MIT, следующая:

```
$ xcode-select --install
$ /bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/in
stall.sh)"
$ brew tap riscv/riscv
$ brew install riscv-tools
```

Далее необходимо добавить в настроечный файл оболочки, например ~/.bashrc, команду дополнения переменной $PATH необходимыми каталогами

```
PATH=$PATH:/usr/local/opt/riscv-gnu-toolchain/bin
```

и установить QEMU:

```
$ brew install qemu
```

Пользователи Windows и macOS могут также установить любой поддерживаемый дистрибутив ОС на базе Linux (Ubuntu, Debian), в т. ч., например, серверную версию Debian для экономии ресурсов, на виртуальную машину. Можно воспользоваться бесплатным ПО для создания и запуска виртуальных машин, например VirtualBox. Следует аккуратно провести установку ОС, установить дополнения гостевой ОС в ней, и настроить общие папки для обмена данными с ОС хоста, следуя инструкциям ПО виртуальной машины.

Получение и запуск xv6

Система xv6 скачивается с официального репозитория

```
git clone https://github.com/mit-pdos/xv6-riscv.git
```

Альтернативно можно рекомендовать создавать собственный форк репозитория для размещения решений задач или воспользоваться предоставленным преподавателем или образовательным учреждением форком. Далее следует перейти в рабочий каталог репозитория.

```
$ cd xv6-riscv
```

Компиляция и запуск операционной системы осуществляется командой

```
[xv6-riscv]$ make qemu
```

При успешной сборке должно вывестись приглашение командной строки оболочки xv6:

```
[xv6-riscv]$ make qemu
# …
# Выводится много информации о процессе сборки
# …
xv6 kernel is booting
hart 2 starting
hart 1 starting
init: starting sh
$ # Это приглашение командной строки xv6
```

Далее можно вводить команды оболочки. Например,

```
$ echo Hello, world
Hello, world
```

или

```
$ ls
.              1 1 1024
..             1 1 1024
README         2 2 2305
cat            2 3 33360
echo           2 4 32208
forktest       2 5 16072
grep           2 6 36776
init           2 7 32664
kill           2 8 32136
ln             2 9 31952
ls             2 10 35456
mkdir          2 11 32200
rm             2 12 32176
sh             2 13 54768
stressfs       2 14 33056
usertests      2 15 181552
grind          2 16 48552
wc             2 17 34328
zombie         2 18 31536
hello          2 19 30976
console        3 20 0
```

и другие. Завершение работы осуществляется с помощью сочетания клавиш Ctrl+A (совместное нажатие) и затем X (Ctrl уже должен быть отпущен):

```
$ QEMU: Terminated
```

```
[xv6-riscv]$ здесь стандартное приглашение оболочки ОС хоста
```

Не следует оставлять xv6 запущенной дольше необходимого (например, в фоне), т. к. xv6 не проводит, а QEMU со своей стороны не реализует политики экономии загрузки процессора, поэтому процесс xv6 будет на 100% загружать используемые ядра процессора ОС хоста, что приводит к высокому энергопотреблению и работе системы охлаждения на полную мощность.

В процессе компиляции xv6 создается ряд файлов, на которые следует обратить внимание:

- `fs.img` в генеральном каталоге — файл-образ виртуального диска, содержащий ФС xv6 со всеми файлами ОС (кроме файла ядра, который загружается QEMU напрямую с ФС хоста);
- двоичные исполняемые файлы формата ELF: `kernel/kernel` — файл ядра xv6, `user/_*` (`user/_cat`, `user/_echo` и др.) — файлы приложений пользовательского пространства (помещаются в корневой каталог `fs.img` без символа-подчеркивания в качестве префикса — `/cat`, `/user` и т. д.);
- файлы дампа дизассемблирования исполняемого кода ядра (`kernel/kernel.asm`) и приложений пользовательского пространства (`user/cat.asm`, `user/echo.asm` и др.);
- соответствующие объектные модули и файлы адресов экспортируемых символов.

В xv6 не используются разделяемые библиотеки, поэтому все файлы приложений фактически компонуются только статически с объектными модулями `ulib.o`, `usys.o`, `printf.o` и `umalloc.o`: дизассемблирование каждого двоичного файла каждого приложения пользовательского пространства показывает, что они (файлы) содержат код всех функций, размещенных в этих объектных модулях. Чтение дампа дизассемблирования приложений может быть полезно в изучении работы компилятора и компоновщика, структуры ассемблерного и исполняемого кода, а также использование машинных инструкций на архитектуре RISC-V.

Следует обратить внимание, что файл `fs.img` генерируется каждый раз при запуске ОС (команда make), поэтому все изменения, сделанные в нем средствами xv6, де-факто носят временный характер, хотя при работе с реальными ОС диск должен рассматриваться как накопитель постоянных данных. Это вызвано, в частности, учебным характером ОС xv6 и необходимостью в частой пересборке образа виртуального диска при подготовке, тестировании и отладке решений задач. Запись данных (программ) в файл-образ диска осу-

ществляется средствами Makefile, штатных средств для извлечения созданных в процессе работы приложений xv6 данных не предусмотрено.

Запуск xv6 в отладчике

Для запуска RISC-V в отладчике следует запустить два процесса — собственно xv6 и отладчика в интерактивном режиме. Сделать это можно в двух отдельных окнах (можно также воспользоваться многооконным графическим эмулятором терминала или текстовым терминальным мультиплексором, например tmux). Запуск xv6 в режиме отладки осуществляется командой

```
make qemu-gdb
```

после чего виртуальная машина войдет в режим ожидания соединения (сетевого соединения посредством прослуширования порта на локальном сетевом адресе) отладчика:

```
[xv6-riscv]$ make qemu-gdb
sed "s/:1234/:26001/" < .gdbinit.tmpl-riscv > .gdbinit
*** Now run 'gdb' in another window.
qemu-system-riscv64 -machine virt -bios none -kernel
kernel/kernel -m 128M -smp 3 -nographic -global virtio-
mmio.force-legacy=false -drive
file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-
device,drive=x0,bus=virtio-mmio-bus.0 -S -gdb tcp::26001
```

При этом создается инициализационный файл отладчика .gdbinit, регулирующий его работу. Данный файл должен находиться в каталоге запуска отладчика gdb, чтобы быть прочитанным им автоматически (т. е. важно запускать gdb именно из генерального каталога исходного кода xv6). Однако в большинстве систем такая автозагрузка по умолчанию отключена из соображений безопасности, поэтому перед первым запуском отладчика следует добавить генерируемый инициализационный файл в список доверительных. Это делается путем добавления в настроечный файл gdb — ~/config/gdb/gdbinit строки

```
add-auto-load-safe-path /home/username/path/to/xv6-riscv/.gdbinit
```

в которой следует указать актуальный путь к инициализационному файлу (конфигурационный файл и дерево каталогов до него нужно создать, если они не существуют). После этого можно вызывать отладчик стандартным образом

```
[xv6-riscv]$ gdb
# Здесь выводится информация об отладчике и советы
0x0000000000001000 in ?? ()
```

```
(gdb)
```

Альтернативно можно прочитывать инициализационный файл вручную с помощью команды отладчика source при каждом его запуске:

```
source .gdbinit
```

Далее можно проводить работу в отладчике (настройку точек останова и др.), продолжать загрузку и работу ОС командой cont и выполнять другие стандартные действия по отладке. При этом работа с ОС (оболочкой, приложении, консольный ввод/вывод и т. д.) осуществляется в процессе виртуальной машины.

Отладочные символы кода ядра отладчик загружает автоматически. Загрузка отладочных символов программ пользовательского пространства должна осуществляться вручную. Например, для получения возможности анализа кода утилиты cat — задания точек останова, просмотра стека вызовов и т. д. — следует набрать

```
file user/_cat
```

то есть передать именно исполняемый файл.

Файл `.gdbinit` создается на основе файла-шаблона `.gdbinit.tmpl-riscv` средствами `Makefile`. При неработоспособности отладчика на некоторых дистрибутивах Linux может потребоваться изменение двух последних файлов. Редактировать файл `.gdbinit` не рекомендуется, так как его содержимое перезаписывается утилитой MAKE каждый раз при запуске ОС в отладочном режиме. Также в шаблон можно добавить команды автоматической загрузки отладочной информации необходимых пользовательских приложений.

Создание пользовательских приложений

Для создания пользовательского приложения в xv6 следует выполнить два действия:

- создать файл исходного кода приложения в каталоге `user`;
- добавить приложение в правила сборки `Makefile`.

Рассмотрим простейшее приложение, выводящее сообщение «Hello, world». Для этого необходимо создать файл `user/hello.c` со следующим содержанием:

```
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"
```

```
int
main(int argc, char *argv[])
{
    printf("Hello, world\n");
    exit (0);
}
```

Затем добавить его в `Makefile` к имеющемуся списку приложений (добавление показано жирным шрифтом):

```
UPROGS=\
      $U/_cat\
      $U/_echo\
      $U/_forktest\
      $U/_grep\
      $U/_init\
      $U/_kill\
      $U/_ln\
      $U/_ls\
      $U/_mkdir\
      $U/_rm\
      $U/_sh\
      $U/_stressfs\
      $U/_usertests\
      $U/_grind\
      $U/_wc\
      $U/_zombie\
      $U/_hello
```

После сборки и запуска системы программа может быть запущена из оболочки:

```
[xv6-riscv]$ make qemu
# …
xv6 kernel is booting
hart 2 starting
hart 1 starting
init: starting sh
$ hello
Hello, world
$
```

В ветке hello репозитория с примерами данного пособия, приводится более сложный вариант программы hello-world. Рассмотрим код функции main с комментариями. Читатели могут изучать этот и другие примеры, определяя

все изменения файлов, которые произведены для создания соответствующих приложений и компонентов ядра, и их последующего включения в ОС, путем сравнения посредством git-diff ветви примера и оригинальной (riscv) ветви.

```
// Обычный printf вывод. В xv6 эта функция
// ничего не возвращает, ошибку не проверить
printf("Hello, world (printf)\n");

// Вывод с помощью системного вызова write
// Количество выводимых байтов на 1 меньше, т. к.
// выводить символ '\0' не нужно
char msg[] = "Hello, world (write)\n";
int len = sizeof(msg) – 1;
int ret = write(1, msg, len); // 1 – дескриптор stdout

// Проверка, вывелись ли все байты
if (ret == len)
    exit (0); // Да, выход (успех)
else
{
    // Вывод сообщения об ошибке в поток ошибок (2)
    // Здесь уже не проверяем ошибку
    write (2, "Write failure (stdout)\n", 23);
    exit (1); // Выход (неудача)
}
```

В коде проводится проверка ошибки ввода/вывода в стандартный поток вывода. Хотя конкретно в этом примере такая ситуация маловероятна и проверка может быть избыточна, при реализации приложений и, особенно, компонентов ядра, следует тщательно проверять все возникающие ошибки, корректно их обрабатывать, не допускать неопределенное поведение. О возникших ошибках следует сообщать пользователю корректным образом — сообщением в поток ошибок и выставлением ненулевого кода возврата. Данный пример иллюстрирует способы это сделать.

Проверить, что код вывода сообщений об ошибке в этом примере действительно выполняется, можно, например, подменив условие ошибки. В том, что потоки ошибок и вывода в xv6 разделены, можно убедиться с помощью их перенаправления:

```
$ hello
Hello, world (printf)
Hello, world (write)
Write failure (stdout)
```

```
$ hello >out
Write failure (stdout)
$ hello 2>err
Hello, world (printf)
Hello, world (write)
$ cat out
Hello, world (printf)
Hello, world (write)
$ cat err
Write failure (stdout)
$
```

Проверить код возврата приложения средствами оболочки не представляется возможным.

В ОС xv6 прозрачным образом допускается включение кода приложений не только на Си, но и на Ассемблере. Для этого следует создать файл с расширением `S` (заглавная буква). В ветке hello-rv приведен пример следующего файла `user/hello.S`, выполняющую ту же работу, что и `hello.c` из предыдущего примера.

```
# Начало сегмента данных
.data
# Определим три 0-терминированных строковых константы
# Для printf, write и сообщения об ошибке
    msg_fmt: .asciz "Hello, world (printf)\n"
    msg_wrt: .asciz "Hello, world (write)\n"
    msg_err: .asciz "Write failure (stdout)\n"
# Длина сообщения (write)
#define MSG_LEN 21

# Начало текстового сегмента (кода приложения)
.text
# Глобальная (экспортируемая) метка
# Важно: main, не _main, не _start — отличается от Linux
.global main

# Начало функции main
main:
# Вызов printf, адрес строка формата — в регистр a0
    la a0, msg_fmt
    call printf
```

```
# Вызов write: 3 аргумента в a0, a1 и a2
# Номер системного вызова — в a7
    li a7, 16       # Номер системного вызова (write)
    li a0, 1        # Поток вывода (stdout)
    la a1, msg_wrt  # Адрес буфера
    li a2, MSG_LEN  # Количество выводимых байтов
    ecall           # Системный вызов
# Проверка ошибки — код возврата сейчас в a0
    li t0, MSG_LEN      # То, что должно вернуться
                        # сохраняется в регистре t0
    bne a0, t0, failure # Условный переход
                        # (если значения не равны)

# Если остались здесь — ошибки нет
    li a7, 2        # Номер системного вызова (exit)
    li a0, 0        # Код возврата (успех)
    ecall
failure:            # Если ошибка
    li a7, 16       # Номер системного вызова (write)
    li a0, 2        # Поток ошибок (stderr)
    la a1, msg_err  # Адрес буфера
    li a2, 23       # Количество байтов
    ecall
    li a7, 2        # Вызов exit
    li a0, 1        # С кодом 1 (ошибка)
    ecall
```

Номера системных вызовов xv6 *отличаются от таковых в других ОС, в частности, в* Linux, *и могут быть найдены в файле* `kernel/sys_call.h`. Включение приложения в `Makefile` производится такой же строкой как и приложения, написанного на Си — добавкой в список `UPROGS` целевого файла `$U/_hello`. На каком именно языке написан исходный код приложения — на Си или Ассемблере — определяется автоматически при сборке. *Перед переключением с ветки на ветку может понадобиться провести очистку репозитория от файлов сборки*

```
[xv6-riscv]$ make clean
```

так как в противном случае может возникнуть конфликт названий и языков приложений. Это же может касается изменения состава и языка программирования файлов исходного кода xv6 в рамках одной ветки.

При написании пользовательских приложений предполагается, что каждое из них состоит из *одного файла исходного кода*. По умолчанию `Makefile`

производит компоновку только с объектными модулями библиотеки ulib. *Разбиение кода приложений на отдельные файлы исходного кода (единицы трансляции) и использование совместных заголовочных файлов требует дополнительной модификации* `Makefile` *для ручной настройки соответствующих правил сборки, и не рекомендуется без острой необходимости*. Альтернативно можно использовать inline-функции и функциональные макросы, определяемые в общих заголовочных файлах.

Создание системных вызовов и компонентов ядра

Для добавления в xv6 системных вызовов необходимо выполнить следующие действия.

- Добавить в файл `kernel/syscall.h` константу с номером системного вызова, начинающуюся с префикса `SYS_` (номер системного вызова должен быть уникальным), например:

```
// …
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
#define SYS_hello  22
```

- Добавить прототип функции ядра обработки системного вызова в файл `kernel/syscall.c`. Функция не должна иметь аргументов и должна иметь возвращаемое значение типа uint64. Рекомендуется использовать принятый в xv6 префикс `sys_` для имен таких функций, например

```
// …
extern uint64 sys_link(void);
extern uint64 sys_mkdir(void);
extern uint64 sys_close(void);
extern uint64 sys_hello(void);
```

- В том же файле добавить указатель на эту функцию в ***таблицу системных вызовов***, представленную массивом `syscalls`:

```
static uint64 (*syscalls[])(void) = {
// …
 [SYS_link]    sys_link,
 [SYS_mkdir]   sys_mkdir,
 [SYS_close]   sys_close,
 [SYS_hello]   sys_hello,
};
```

- Реализовать функцию с префиксом `sys_`. Сделать это можно в одном из файлов исходного кода ядра. Однако надо учесть, что файлы с функциями обработки системного вызова в xv6 начинаются с префикса `sys`, а файлы без данного префикса используются для функций ядра, вызываемых из системных вызовов и других компонентов ядра. *Функции ядра, осуществляющие обработку системного вызова (начинающиеся с* `sys_`*), не могут быть вызваны из других функций ядра, так как передача аргументов в них производится особым образом — они должны получить их не из ядра, а из пользовательского пространства*. Разместить тела этих функций можно в файле, реализующем ту же часть функциональности, что и добавляемый системный вызов, или в отдельном файле исходного кода ядра, например, `kernel/syshello.c`:

```
#include "types.h"
#include "riscv.h"
#include "defs.h"
uint64 sys_hello(void)
{
    printf("Hello, world (system)\n");
    return 0;
}
```

  Данный системный вызов просто выводит сообщение «Hello, world» на консоль нехарактерным для реальных ОС, но доступным в xv6 способом — с помощью функции printf — и всегда возвращает 0.

- Добавить прототип оберточной функции системного вызова в пользовательском пространстве, в файл `user/user.h`, например:

```
// …
char* sbrk(int);
int sleep(int);
int uptime(void);
int hello(void);
```

  В данном прототипе у функции должны указываться аргументы, которые будут переданы системному вызову. В данном случае системный вызов параметров не имеет, поэтому используется `void`.

- Обеспечить генерацию ассемблерного кода тела данной оберточной функции, добавив строку в файл `user/usys.pl`, например:

```
# …
entry("sbrk");
```

```
entry("sleep");
entry("uptime");
entry("hello");
```

- После этого системный вызов можно использовать посредством оберточной функции в приложениях пользовательского пространства, например, создать файл `user/hello.c`:

```
#include "kernel/types.h"
#include "user/user.h"

int
main(int argc, char *argv[])
{
  hello();
  exit(0);
}
```

- Далее необходимо добавить новый файл исходного кода ядра и приложение пользовательского пространства в `Makefile` к спискам OBJS и UPROGS соответственно:

```
OBJS = \
 # …
 $K/sysfile.o \
 $K/kernelvec.o \
 $K/plic.o \
 $K/virtio_disk.o \
 $K/syshello.o

# …

UPROGS=\
        # …
        $U/_grind\
        $U/_wc\
        $U/_zombie\
        $U/_hello
```

- После этого можно запускать указанное приложение.

```
[xv6-riscv]$ make qemu

# …

xv6 kernel is booting
hart 2 starting
hart 1 starting
init: starting sh
```

```
$ hello
Hello, world (system)
$
```

Код указанного примера представлен в ветке sys_hello репозитория с примерами.

Код системного вызова может быть реализован и на Ассемблере. Например, вызов sys_hello можно реализовать в файле `syshello.S` (все остальные действия остаются неизменными):

```
# Данные (строка формата и выводимая строка)
fmt: .asciz "%s"
msg: .asciz "Hello, world (system)\n"

# Метка (глобальная) функции
.globl sys_hello
sys_hello:

    # Сохранение в стеке регистров, которые изменяют
    # значения при работе данной функции, но должны
    # восстановить их по завершении
    # (callee-saved) – ra и s1
    addi sp, sp, -16
    sd ra, 0(sp)
    sd s1, 8(sp)

    # Загрузка аргумента фуннции
    la a0, fmt
    la a1, msg

    # Загрузка адреса функции printf: прямой вызов
    # с помощью immediate-инструкции может быть
    # невозможен из-за ограничений на длину значения
    # адреса: код ядра сравнительно большой
    la s1, printf

    # Вызов функции по адресу (изменение ra)
    jalr ra, s1, 0

    # Восстановление значений регистров
    ld ra, 0(sp)
    ld s1, 8(sp)
    addi sp, sp, 16

    # Выход из функции (sys_hello)
    ret
```

Разумеется, в данном случае функция printf позволяет вывести строку без использования `"%s"`. Использовать строку данных, пришедших от пользо-

вателя, в качестве первого аргумента printf нельзя, так как она может содержать символы форматирования, интерпретация которых приведет к выводу непредсказуемых данных и потенциально может служить источником уязвимости. В данном примере используется фиксированная строка, поэтому такой проблемы не возникнет, а код с использованием "%s" приводится как пример хорошего подхода и ради иллюстрации многопараметрического вызова printf. Этот пример приведен в ветке sys_hello_rv.

Поскольку при переходе в режим ядра происходит переключение на контекст ядра, в т. ч. изменяются значения регистров и подгружается таблица страниц ядра, прямая передача аргументов в функцию-обработчик системного вызова в ядре (которая указывается в таблице системных вызовов — с префиксом sys) невозможна. Вместо этого значения регистров сохраняются в структуре trapframe процесса и должны быть извлечены оттуда специальными функциями. Например, для реализации системного вызова с двумя целочисленными аргументами следует создавать оберточную функцию с таким прототипом (файл user/user.h):

```
int add (int, int);
```

а функция, обрабатывающая системный вызов со стороны ядра, должна действовать подобным образом:

```
uint64 sys_add (void)
{
   int n, m;
   argint(0, &n);
   argint(1, &m);
   return n + m;
}
```

Данный системный вызов возвращает сумму двух аргументов (разумеется, для сложения двух чисел не требуется привилегий режима супервизора, и такой системный вызов, как и предыдущий, приводится чисто в качестве иллюстрации процесса их создания). Воспользоваться системным вызовом можно обычным образом (в пользовательском приложении):

```
int sum = add (p, q);
```

В ситуации, когда в системный вызов или из него требуется передать не целочисленное значение, а более сложное, например, строковый буфер, структуру или массив целых чисел, собственно аргументом системного вызова является адрес этого значения в памяти. Он извлекается с помощью функции argaddr. Так как данный адрес является адресом в виртуальном адресном пространстве процесса, разыменовывать его в пространстве ядра нельзя, это

приведет к доступу к совершенно другому физическому адресу: требуется копирование данных из пользовательского пространства в пространство ядра (входных данных системного выхода) с помощью функций copyin и argstr (для строк) и обратно (выходных данных системного вызова) с помощью функции copyout.

Эти функции осуществляют копирование только если оно корректно, т. е. указанный диапазон адресов действительно принадлежит виртуальному адресному пространству процесса, сделавшего системный вызов, в противном случае возвращают ошибку. Проверка наличия ошибки копирования данных при обработке системного вызова обязательна. При обнаружении такой ошибки системный вызов должен вернуть пользовательскому приложению соответствующий код (обычно отрицательное число). Попытка чтения или записи по адресу, не принадлежащему процессу, может привести к нарушению изоляции и появлению уязвимостей в безопасности.

В ветке sys_args рассматривается системный вызов условно безопасного копирования строк. В пользовательском пространстве используется следующая функция

```
int scpy(char *dst, const char *src, int size);
```

Здесь аргумент `dst` — буфер назначения, `size` — его размер (предельное число копируемых байтов включая терминирующий ноль), `str` — исходная строка. Системный вызов реализован следующим образом:

```
uint64 sys_scpy(void)
{
    // Получение адреса назначения, тип uint64,
    // а не void *, т. к. это виртуальный адрес
    uint64 dst;
    argaddr(0, &dst);
    // Создание буфера в ядре, фиксированного размера
    // Самая простая реализация: строки большей
    // или равной длины не копируются
    char buf[BUF_LEN];

    // Извлечение входной строки в буфер с помощью argstr
    // Функция вернет длину строки или отрицательное
    // значение при ошибке
    int len;
    len = argstr(1, buf, BUF_LEN);

    // Возвращаем код ошибки -1,
```

```
    // если копирование не удалось
    if (len < 0) return -1;

    // Получаем размер входного буфера
    int size;
    argint(2, &size);

    // Проверяем, что в нем достаточно места для строки
    // длина len и символа '\0', иначе — код ошибки -1
    if (len + 1 > size) return -2;
    // Попытка копирования в буфер назначения len + 1
    // байтов, вернет отрицательное значение, если ошибка
    int ret = copyout(myproc()->pagetable,
                      dst, buf, len + 1);
    // При ошибке копирования выходных данных — код -3
    if (ret < 0) return -3;
    // Если все в порядке — возвращаем 0
    return 0;
}
```

В реализации используется буфер в стеке ядра фиксированного размера, который должен вместить строку целиком. Следует учесть, что данный буфер должен уместиться в небольшом стеке ядра, что накладывает сильное ограничение на размер копируемой строки. Реализация более высокого качества может использовать копирование в цикле порциями, не превышающими размер буфера. Ядра реальных ОС допускают также выделение буфера динамически из кучи (в xv6 соответствующих функций ядра не предусмотрено). Однако копирование порциями посредством статического стекового буфера предпочтительнее в силу того, что динамическое выделение памяти может не сработать и потребует дополнительных расходов. При этом стековый буфер не должен быть большим, т. к. стек ядра процесса мал. Использование глобального буфера потребует синхронизацию (использование блокировки) в качестве защиты от гонки при параллельном использовании системного вызова разными процессами. Это создаст дополнительную задержку.

Использование системного вызова осуществляется по схеме

```
// Исходная строка, буфер назначения и код возврата
char str[] = "Hello, world";
char buf[BUF_SIZE];
int ret;
// т. к. при ошибке системный вызов не изменяет буфер,
// обеспечиваем в нем пустую строку
```

```
buf[0] = '\0';

// Попытка копирования
ret = scpy (buf, str, BUF_SIZE);
// Проверка кода возврата
printf(ret == 0 ? "Success" : "Copy error");
// Вывод кода возврата и содержимого буфера
printf("\treturn code %d\t"
       "buffer content: '%s'\n",
      ret, buf
     );
```

В ветке sys_args представлено несколько тестов, иллюстрирующих различные ситуации — использование заведомо некорректного входного и выходного адреса, слишком малого и большого размера буфера, а также использование NULL (0) в качестве адреса. В качестве эксперимента можно также превысить размер буфера ядра.

Исследуя поведение функции, можно заметить, что ядро xv6 допускает чтение и запись по нулевому адресу пользовательского пространства с непредсказуемым поведением. Поэтому для обеспечения защиты от такой ситуации требуется дополнительная проверка в коде ядра. В данном случае при использовании только argstr это затруднительно, поэтому следует воспользоваться функцией argaddr для получения адреса входной строки (и, возможно, вызвать copyin вместо argstr).

Кроме того, функции copyin, argstr и copyout не защищают от выхода за границы массива — только от выхода за границы адресного пространства процесса. Таким образом, *ядро не может обеспечить полную безопасность программы и защитить от непредсказуемого поведения из-за перезаписи данных или кода приложения, защита предоставляется только от обращения по адресу, не принадлежащему процессу*. Но эти проверки все равно являются обязательными при реализации всех системных вызовов по соображениям безопасности.

Более качественная реализация данного системного вызова должна использовать символические константы для кодов ошибок, доступные как из пространства ядра, так и из пользовательского пространства. В xv6 не применяется такой подход, а в Linux системный вызов возвращает отрицательный код ошибки, абсолютное значение которого потом присваивается оберточной функцией в переменную errno. Например, можно добавить в ядро xv6 заголовочный файл `errno.h`:

```
#define ESUCCESS 0 // No error
#define ENULL    1 // NULL userspace address
#define EFAULT   2 // Bad userspace address
#define ENOMEM   3 // Not enough kernel memory
#define ESIZE    4 // Bad buffer size
```

Тогда системный вызов примет вид:

```
uint64 sys_scpy(void)
{
    // Получаем адрес назначения, проверяем на NULL
    uint64 dst;
    argaddr(0, &dst);
    if (dst == 0) return -ENULL;

    // Аналогично с адресом источника
    uint64 src;
    argaddr(1, &src);
    if (src == 0) return -ENULL;

    // Копируем входную стоку в буфер buf, побайтово
    // Следя за статусом копирования ret и границей
    // буфера (указатель d пробегает буфер записи)
    char buf[BUF_LEN];
    int ret;
    char *d = buf;
    do
    {
        ret = copyin(myproc()->pagetable, d, src++, 1);
    } while (ret >= 0 && *d++ != '\0'
                      && d < buf + BUF_LEN);

    // Ошибка копирования
    if (ret < 0) return -EFAULT;

    // Выход за границы буфера ядра
    if (d >= buf + BUF_LEN) return -ENOMEM;

    // Успех копирования, вычисляем число
    // скопированных байтов (длина строки +1)
    int len = d - buf;

    // Размер выходного буфера
    int size;
    argint(2, &size);

    // Ошибка, буфер слишком мал
    if (len + 1 > size) return -ESIZE;
```

```
    // Попытка копирования в выходной буфер
    ret = copyout(myproc()->pagetable,
                  dst, buf, len);
    if (ret < 0) return -EFAULT;
    // Успех
    return 0;
}
```

Далее в пользовательском пространстве можно добавить функцию печати текстового представления кода ошибки, например в `user.h` задать прототип

```
void perror(int errno);
```

В функцию будет передаваться положительный код ошибки (в Linux вместо этого автоматически выставляется и используется значение глобальной переменной errno). В файле `printf.c` можно создать таблицу текстовых представлений ошибок и код функции perror, выводящий эти сообщения в поток ошибок (более изящная реализация должна включать функцию получения и замены текстового кода ошибки в приложении):

```
char *errmsg[] = {
  [ESUCCESS] "No error",
  [ENULL]    "NULL userspace address",
  [EFAULT]   "Bad userspace address",
  [ENOMEM]   "Not enough kernel memory",
  [ESIZE]    "Bad buffer size"
};
void perror(int errno)
{
  fprintf(2, errmsg[errno]);
  fprintf(2, "\n");
}
```

Далее тестовый код системного вызова приобретает вид

```
buf[0] = '\0';
ret = scpy (buf, str, BUF_SIZE);
perror(-ret);
printf("return code %d\tbuffer content: '%s'\n",
       ret, buf
      );
```

Этот вариант реализации приведен в ветке sys_args_err. *Читателю рекомендуется внимательно изучить изменения, сделанные в каждой ветке, и самостоятельно исследовать приведенные и собственноручно созданные тесты поведения программ и системных вызовов.*

Требования к качеству решений и программному коду

При решении задач пособия (и вообще написании кода для ОС) помимо следования стандартным принципам написания хорошего кода (в части оформления и комментирования) следует учитывать ряд требований и рекомендаций.

- Следует избегать ненужной сложности и стремиться к простому для понимания, компактному и прозрачному решению, как с точки зрения применяемых приемов, так и в части самого кода. Если решение остается простым, то в нем легче не допустить ошибку, его проверить и добиться лучшего качества работы. Этот подход известен как принцип KISS (первоначальная расшифровка — «*Keep it simple, stupid*» — «*Делай проще, тупица*»; встречаются и другие, более нейтральные с этической точки зрения и точнее конкретизирующие суть расшифровки, такие, как «*keep it small and simple*» — «*сохраняй это маленьким и простым*» и «*keep it simple and straightforward*» — «*оставляй это простым и прямолинейным*», указывающие на необходимость стремиться к короткому, простому и напрямую понятному решению).
- Следует проверять и обрабатывать все возникающие во время работы программ и ОС ошибки (ввода/вывода, некорректные входные данные и т. д.). На уровне пользовательских приложений проверять все возможные варианты ответов от ОС (системных вызовов), а также (там где это возможно) — корректность вводимых пользователем входных данных и аргументов командной строки. *При обнаружении ошибки следует оповещать пользователя соответствующим сообщением. Это сообщение должно позволить пользователю понять причину ошибки, и как ее можно исправить*. Сообщения об ошибках должны выводиться в поток ошибок. Об ошибках, обнаруженных при обработке системного вызова, должно быть сообщено приложению пользовательского пространства соответствующим кодом возврата. Ошибки, возникающие в ядре асинхронно (при обработке аппаратных прерываний) или в следствие исключений, также должны быть обработаны, сообщение о них должно быть передано в пользовательское пространство (если возможно).

  Стратегия xv6 предполагает выдачу таких сообщений об ошибке на консоль с указанием информации о значении регистров, содержащих адрес инструкции (sepc) и причину ловушки (scause). Если ошибка

произошла в ядре, производится завершение работы ОС с сообщением о панике ядра (***kernel panic***). Если ошибка произошла в пользовательском процессе, процесс завершается принудительно (kill). В реальных ОС (например, в ядре Linux) в таких случаях предполагается передача сообщения в журнал диагностических сообщений ядра, завершение процесса или отключение устройства, вызвавшего ошибку. Остановка работы ядра Linux (*kernel panic*) также производится при фатальной ошибке, когда ядро не может продолжить работу и изолировать сбой.

- Не допускать ошибок программирования, особенно в пространстве ядра. Ни в коем случае не следует полагаться на неопределенное поведение, в т. ч. некорректное обращение к памяти и другие не соответствующие стандарту приемы, игнорировать выявленные ядром или функциями библиотеки ошибки и т. д. Компилятор gcc позволяет выявить многие ошибки такого рода, выводя соответствующие предупреждения компилятора. Сборка xv6 настроена таким образом, чтобы все предупреждения компилятора считались ошибками. Это правильный подход, так как игнорировать предупреждения компилятора нельзя, но он не позволяет выявить все возможные ошибки программирования.

  Следует отметить, что *если ошибки программирования, допущенные в приложениях пользовательского пространства, обычно сказываются только на отдельном приложении и отдельном пользователе системы, то ошибки программирования, присутствующие в пространстве ядра, могут повлечь фатальный сбой всей системы, а также появление уязвимостей в безопасности ОС (для многопользовательских систем)*. Поэтому программирование в пространстве ядра требует особой внимательности и осторожности. Следует также избегать утечек памяти. В ядрах ОС, поддерживающих динамическое выделение памяти, недопущение утечек особенно важно, так как они приведут к напрасному накапливающемуся расходу памяти, который рано или поздно приведет к невозможности продолжения работы системы.

- Обязательно пользоваться механизмами синхронизации при доступе к общим аппаратным и программным ресурсам ядра. Ядро работает в режиме и одновременной и виртуальной многопоточности. В частности, один и тот же системный вызов может быть сделан одновременно несколькими процессами — во время исполнения кода системного

вызова может произойти переключение контекста процесса по таймеру на другой процесс, который сделает тот же самый (или другой) системный вызов. Все глобальные переменные и структуры данных, предполагающие совместное использование, должны быть защищены специфической блокировкой. При обработке прерываний должна использоваться только спин-блокирвка, в других местах при потенциально заметном ожидании следует использовать спящую блокировку. При использовании вложенных блокировок нельзя допускать взаимоблокировок, необходимо устанавливать порядок захвата блокировки ресурсов и следовать ему при захвате во всех местах кода, а освобождение блокировок проводить в обратном порядке.

- Снабжать код тестовыми программами (утилитами пользовательского пространства), проверяющими работоспособность приложений и системных компонентов. Насколько это возможно, делать автоматические тесты. Тесты должны подбираться таким образом, чтобы были исполнены все ветви кода (все варианты условий), рассмотрены все возможные варианты исхода и т. д.

При программировании компонентов ядра реальных ОС следует также внимательно заботиться об эффективности используемых алгоритмов и конструкций кода, т. е. качественно оптимизировать работу ядра. От эффективности работы ядра сильно зависит производительность приложений и компьютера в целом. При разработке учебной ОС xv6 данный фактор также следует иметь в виду, но следование принципу KISS и недопущение ошибок программирования являются первичными и более критичными требованиями. Даже в реальных системах оптимизация может рассматриваться как процесс этапа переработки и улучшения кода, а при решении учебных задач этот этап может быть частично опущен. Тем не менее в очевидных местах следует всегда отдавать предпочтение более эффективным алгоритмам и приемам уже на этапе написания кода — сокращать количество проходов массива, исключать повторяющиеся и избыточные вычисления, использовать арифметику указателей, отдавать предпочтение асимптотически лучшим алгоритмам.

§4.2. Задачи на разработку приложений пользовательского пространства

Задачи данного раздела предполагают знакомство с xv6, ее кодом и имеющимися системными вызовами (во многом схожими с системными функциями POSIX). Их решения могут быть выполнены на языке Си или на Ассемблере RISC-V как с использованием функций библиотеки Си пользова-

тельского пространства xv6 (ulib), так и без них, т. е. только с использованием системных вызовов.

Знакомство с xv6

Лабораторная №1: ознакомительная и подготовительная

1. Установить необходимое ПО и скачать исходный код xv6. Убедиться, что ОС xv6 запускается на виртуальной машине QEMU, в ней запускаются имеющиеся пользовательские приложения, работа системы корректно завершается.
2. Убедиться, что xv6 запускается в отладчике. Провести через отладчик одно из пользовательских приложений системы, например, программу echo или cat: подгрузить отладочные символы, установить точку останова на одну из строк кода, выполнить пошаговое исполнение операторов, провести наблюдение за переменными.
3. Написать или исполнить готовую (из ветки примеров к данной книге) пользовательскую утилиту, выводящую на экран сообщение «Hello, world», используя printf и write.
4. Написать пользовательскую утилиту для вычисления суммы двух введенных чисел. Утилита вводит с клавиатуры строку, содержащую два числа, разделенных пробелом, затем конвертирует оба строковых представления числа в данные типа int (с помощью atoi — только положительные числа, или вручную) и выводит их сумму в терминал, используя форматированный вывод printf. Строку вводить с помощью функции gets в буфер адекватного размера (вычислить его исходя из длины десятичного представления 32-битных чисел).

   Учесть, что ввод строки пользователем может быть завершен не только нажатием клавиши Enter, но и из-за достижения конца файла. В первом случае в строку помещается символ перевода строки (`'\n'`), во втором он будет отсутствовать. Конец файла может быть введен с клавиатуры с помощью комбинации Ctrl+D или быть достигнут при перенаправлении ввода из текстового файла: в конце файла символ перевод строки может как присутствовать, так и отсутствовать.

   Исследовать строку на возможные ошибки формата ввода: некорректные символы в строке, отсутствие одного или обоих чисел, присутствие более чем одного числа и т. п. При обнаружении ошибки вывести сообщение в поток ошибок.

Ввод и вывод

Лабораторная №2: файловый ввод и вывод

1. Разработать утилиту, которая для каждой строки заданного входного текстового файла вычисляет ее порядковый номер и длину. Файл передается в качестве параметра командной строки. Если параметр отсутствует, используется стандартный ввод. Результат выводится на стандартный вывод: для каждой строки исходного файла выводятся два числа, разделенные пробелом — номер и количество символов, исключая символ перевода строки (`'\n'`). Учесть, что в конце файла символ перевода строки может отсутствовать. Если символ перевода строки является последним байтом в файле, то возникающая в конце файла пустая строка игнорируется. Ввод осуществлять с помощью системного вызова read в буфер адекватного размера (обычно для оптимальности рекомендуется использовать размер буфера, кратный размеру дискового блока, который в xv6 составляет 1024 байта). Учесть, что даже если в файле остались данные, реальное количество введенных байтов может быть меньше размера буфера (стандарт допускает, что read прочитывает от 1 до размера буфера байтов). Обработать все возможные ошибки: некорректный формат аргументов командной строки, ошибки ввода/вывода — открытия и закрытия файла, чтения и т. д.
2. Дополнить утилиту таким образом, чтобы результат выводился в формате: номер строки, собственно строка, длина строки. Выбрать адекватный разделитель: пробел, символ табуляции или другой. Выводить информацию в выходной файл с помощью write, проверяя количество выводимых данных и ошибки записи. Имя выходного файла задается в качестве параметра командной строки, параметр отсутствует (имеется только один параметр — входной файл) — используется стандартный вывод. Если в качестве имени входного или выходного файла задан дефис (`'-'`), то в качестве данного файла используется стандартный ввод или вывод соответственно. Обработать все возможные ошибки: формат аргументов командной строки, ошибки ввода/вывода и т. д.
3. Написать утилиту, генерирующую тестовые текстовые файлы, обладающие заданными свойствами (строки разной случайной длины, в т. ч. превышающей размер буфера из задачи (2), пустые строки в середине файла, с наличием или отсутствием символа перевода строки

в конце файла, пустые файлы — размера 0). Проверить работоспособность утилит (1) и (2) на указанных файлах.

Лабораторная №3: создание функций строкового ввода/вывода

1. Реализовать функцию

```
int fgets (char *buf, int size, int fd);
```

прочитывающую строку из указанного файлового дескриптора в заданный буфер размера `size`. Функция всегда должна записывать в буфер нуль-терминированную строку, максимальное количество прочитываемых байтов должно составлять `size-1`. Функция читает строку из файла до момента достижения символа перевода строки или до достижения конца файла. Символ перевода строки прочитывается из файла, но не включается в строку. Возвращаемое значение функции — код одного из возможных исходов операции:

- строка или ее остаток прочитаны полностью (в т. ч. пустая строка), конец файла не достигнут — в буфере было достаточно места, следующий вызов будет прочитывать новую строку (если она есть);
- строка или ее остаток прочитаны не полностью, в буфере недостаточно места — следующий вызов будет прочитывать остаток строки;
- строка или ее остаток прочитаны полностью, но при чтении достигнут конец файла — в конце файла нет символа перевода строки, в буфере было достаточно места — следующий вызов приведет к получению статуса достижения конца файла;
- строка не прочитана, т. к. немедленно достигнут конец файла — предыдущий вызов прочитал последнюю строку полностью, а в конце файла присутствует символ перевода строки или предыдущий вызов уже сообщил о достижении конца файла;
- в процессе чтения возникла ошибка ввода/вывода.

Коды возврата можно определить как символические константы с помощью #define или enum. Указанная реализация отличается от функции fgets стандартной библиотеки Си в т. ч. по причине отказа от использования глобальной переменной errno. Для большего удобства можно добавить дополнительный выходной параметр (указатель), возвращающий длину реально прочитанной строки. Для ввода использовать read, прочитывая файл посимвольно.

Более оптимальное решение может включать в себя реализацию буферизированного файлового ввода. Для этого каждому открытому файловому дескриптору следует сопоставить буфер некоторого размера и прочитывать в него данные с помощью read, *сохраняя позицию чтения и количество прочитанных в буфер байтов. Это можно сделать, в частности, путем реализации структуры* FILE *и функций для работы с указателем на эту структуру поверх целочисленного файлового дескриптора. После этого можно передавать необходимый объем данных из файлового буфера в целевой буфер с помощью* memcpy. *Этот подход требует учета множества факторов и ситуаций, кроме того создаст конфликт при поочередном использовании буферизированного и небуферизированного ввода/вывода.*

Другой подход мог бы заключаться в использовании позиционного ввода/вывода — при прочтении в буфер назначения символа перевода строки и некоторого количества байтов за ним следует уменьшить позицию в файле на количество избыточно прочитанных байтов с помощью системного вызова lseek. *Однако данный системный вызов не реализован в* xv6 *— предварительно требуется его реализация. При реализации следует учесть, что позиция чтения сохраняется в поле* off *структуры ядра* file, *функция* argfd *позволяет получить эту структуру по номеру файлового дескриптора, а все операции ввода/вывода из* inode *опираются на функции* readi/writei, *изначально предоставляющие позиционный ввод и вывод. Также учесть, что позиционирование возможно только в пределах размера файла, поэтому системный вызов* lseek *должен провести необходимые проверки. Позиционирование, однако, возможно только для регулярных файлов, но не каналов и устройств, поэтому такое решение ограничит область применения функции.*

2. Реализовать функцию

```
char *getline (int fd, int *err);
```

вводящую из заданного файла строку в динамически выделяемый из кучи строковый буфер. При использовании этой функции указанный буфер должен быть освобожден по завершении работы с ним с помощью free. Учесть, что в xv6 отсутствует функция realloc, поэтому увеличение буфера следует организовать вручную, путем создания нового буфера и копирования туда данных исходного буфера. Буфер следует увеличивать не побайтово, а блоками — для сокращения ко-

личества копирований. После прочтения всей строки буфер можно уменьшить до нужного размера, но это приведет к дополнительному копированию данных в памяти и может рассматриваться как избыточное действия. В случае ошибки getline должен вернуть 0 (NULL), при этом нужно освободить ранее выделенную внутри функции память, чтобы не допустить утечку. В `*err` следует записывать код ошибки — невозможность выделить память, ошибка ввода/вывода, достижение конца файла до начала чтения.

Создание и взаимодействие процессов

Лабораторная №4: создание и завершение процессов

1. Написать программу, выполняющую следующие действия.
   - Создается дочерний процесс, который приостанавливается на несколько (10-20) секунд и завершается с кодом возврата, задаваемым в качестве параметра командной строки. Если параметр отсутствует, использовать код возврата 0.
   - Родительский процесс выводит свой идентификатор и идентификатор дочернего процесса, затем ожидает завершения дочернего процесса, после чего выводит идентификатор завершенного процесса и его код возврата, затем сам завершается с нулевым кодом возврата.

   Корректно обработать ошибки создания процесса, отправки сигнала, ожидания завершения дочернего процесса и др.
2. Запустить программу (1) в виде фонового процесса. Убедиться, что созданы два ее процесса с одинаковым именем (используя Ctrl+P для вызова печати списка процессов ядром системы). С помощью утилиты kill уничтожить дочерний процесс, посмотреть на поведение родительского процесса.
3. Написать программу, выполняющую следующие действия.
   - Создается дочерний процесс, который приостанавливается на несколько (3-5) секунд и завершается с кодом возврата, задаваемым в качестве параметра командной строки. Если параметр отсутствует, использовать код возврата 0.
   - Родительский процесс выводит свой идентификатор и идентификатор дочернего процесса, затем с помощью kill убивает дочерний процесс, ожидает его завершения, после чего выводит

идентификатор завершенного процесса и его код возврата, затем сам завершается с нулевым кодом возврата.

4. Написать программу, выполняющую следующие действия.
   - Программа вводит строку — список имен файлов, разделенных пробелами.
   - Создается дочерний процесс. Дочерний процесс передает управление утилите wc, при этом ранее введенные имена файлов передаются в качестве параметров командной строки (нуль-терминированный массив нуль-терминированных строк).
   - Родительский процесс ожидает завершения дочернего процесса, после чего выводит идентификатор завершенного процесса и его код возврата, затем сам завершается с нулевым кодом возврата.

   В программе следует корректно обработать все ошибки ввода, создания процесса, запуска приложения и др. Учесть, что при использовании exec *следует указывать абсолютный путь к запускаемой программе, т. к. исходная утилита не обязана запускаться из того же каталога, что и* wc*. Аккуратно передать массив параметров командной строки в* wc*: нулевой элемент — имя утилиты (в любом удобном представлении), последний — нулевой указать. Учесть, что при успехе* exec *следующий за ним код недостижим, поэтому после вызова следует выводить сообщение об ошибке. Убедиться, что утилита* wc *завершается с различным кодом возврата в зависимости от того, удалось ли проанализировать все переданные файлы или нет (возникла ошибка чтения или не удалось открыть файл по причине его отсутствия).*

5. Написать утилиту, запускающую в дочернем процессе другую программу, имя исполняемого файла которой передается как первый параметр командной строки. Оставшиеся параметры запускаемой программы передаются в качестве остальных аргументов командной строки утилиты. По завершении процесса вывести его код возврата и время работы в тиках (использовать uptime для получения информации о времени запуска и завершения процесса). Корректно обработать ошибки формата командной строки, создания процессов, завершения и т. д.

   Оболочка xv6 *не предусматривает средств для получения кодов возврата запущенных приложений. В качестве дополнительной задачи можно рассмотреть расширение возможностей* sh *таким образом,*

чтобы пользователь мог определить код возврата последней исполненной программы, например, выводя его автоматически по завершении работы или подменяя им сочетание $? *в командной строке (в соответствии с поведением реальной оболочки* POSIX*).*

6. Написать простейшую версию утилиты test. В программу передаются три параметра командной строки: левый операнд, операция и правый операнд. В качестве знака операции можно использовать равенство (операнды равны как строки), неравенство (операнды не равны как строки) и сравнение операндов как целых чисел. Если результат операции истина, молча вернуть (код возврата) 0, если ложь — вернуть 1. При ошибке командной строки вывеси сообщение об ошибке и вернуть 2. Протестировать работу программы с помощью программы (5).

   Оболочка xv6 *не реализует использование переменных и условных конструкций. Это расширение функционала можно рассмотреть как дополнительную задачу.*

Лабораторная №5: использование каналов

1. Написать программу, которая создает дочерний процесс и передает ему через заранее созданный анонимный канал (pipe) все аргументы командной строки, завершая каждый из них символом перевода строки (`'\n'`), закрывает канал по завершении записи. Дочерний процесс получает все данные из канала до конца файла (закрытие конца для записи родительским процессом) и выводит их на стандартный вывод, после чего завершается. Родительский процесс ожидает завершение дочернего процесса и завершается сам.

   Хотя ОС при завершении процесса должна закрыть все открытые файловые дескрипторы, рекомендуется закрывать неиспользуемые концы канала — освобождать ресурсы — как можно раньше. Читающий (дочерний) процесс должен сразу закрыть конец для записи — в противном случае он не получит «конец файла» при закрытии его только родительским процессом (конец файла-канала достигается только если все открытые дескрипторы конца для записи — во всех процессах — закрыты). Кроме того, следует проверять на ошибки при создании процессов, каналов, чтении и записи, в т. ч. вызова close *после записи.*

- Написать программу, которая создает дочерний процесс и передает ему через заранее созданный анонимный канал (pipe) все аргументы командной строки, завершая каждый из них символом перевода стро-

ки (`'\n'`), закрывает канал по завершении записи. Дочерний процесс замещает стандартный ввод (0) выходным концом канала с помощью dup:

```
close(0);
dup(pipefd[0]);
```

после чего канал может быть закрыт

```
close(pipefd[0]);
```

Далее дочерний процесс замещается приложением wc без аргументов. Программа wc должна вывести подсчет строк (количества аргументов командной строки), слов и байтов в них. Родительский процесс ожидает завершение дочернего процесса и завершается сам.

Оболочка sh *в* xv6 *не предусматривает экранирование спецсимволов с помощью символов обратного слеша и кавычек. Поэтому пробелы всегда будут служить разделителем аргументов, т. е. количество строк и количество слов, выводимое в этой задаче утилитой* wc, *будут совпадать. В качестве дополнительного задания можно реализовать функционал экранирования пробелов в оболочке* sh, *модифицировав ее исходный код соответственно.*

§4.3. Задачи на разработку системных вызовов и компонентов ядра

Задачи данного пункта предполагают разработку системных вызовов и необходимых для их работы функций ядра, а также модификацию отдельных структур и функций ядра при необходимости. Кроме этого требуется создание тестовых программ пользовательского пространства. Данные программы могут быть разработаны как на Си, так и на Ассемблере RISC-V. Разработка компонентов (функций) ядра предполагается преимущественно на языке Си, но отдельные элементы также могут разрабатываться на RISC-V или потребовать ассемблерных вставок. Решение некоторых задач на Си потребует знания особенностей архитектуры RISC-V, например, регистров, структуры таблицы страниц и причин ошибок.

Программирование системных вызовов

Лабораторная №6: ознакомительная

1. Добавить в xv6 системный вызов sys_add, который получает на вход два целых числа и возвращает их сумму. Написать тестовую утилиту для проверки работы вызова.
2. Добавить в xv6 системный вызов sys_strlen, который получает на вход нуль-терминированную строку символов и адрес целочисленной переменной для записи результата — длины строки. Возвращаемое значение системного вызова должно соответствовать коду ошибки (0 — успех, различные отрицательные значения — возможные ошибки: некорректный адрес строки, некорректный адрес назначения). Исходную строку прочитать побайтово, без использования argstr/copyinstr и предположений о предельной длине строки (размере буфера ядра). Написать тестовую утилиту для проверки работы вызова.
3. Добавить в xv6 функционал номеров ошибок системных вызовов и сообщений о них — errno.
   - Завести в ядре файл `kernel/errno.h`, который будет содержать define-константы кодов ошибок (положительные или отрицательные). Системные вызовы при ошибке должны возвращать отрицательный код ошибки.
   - Добавить в библиотеку ulib глобальную таблицу (массив) текстовых сообщений об ошибках, индекс будет соответствовать положительному коду ошибки (его абсолютному значению), определенному в `errno.h`. Добавить функции получения и модификации текстового сообщения об ошибке по его номеру (положительному или отрицательному — как определено в `errno.h`). Добавить функцию

     ```
     void perror(const char *str, int errno);
     ```

     выводящую сообщение str и текстовое представление об ошибке из таблицы.
   - Реализовать сообщения об ошибках в задаче (2) с использованием данной задачи.

   Дополнительно можно реализовать функционал передачи информации об ошибке с использованием классического подхода — глобальной переменной errno. *Завести такую переменную в* `ulib.c`. *Добиться того, чтобы генерируемый с помощью* `usys.pl` *код оберточных функций системных вызовов проверял их возвращаемое значение и записывал это значение в глобальную переменную* errno *по абсолютной величине (в случае ошибки — получения отрицательного резуль-*

тата), а при неотрицательном результате в errno *должен записываться 0 (нет ошибки).*

Учесть, что в реальной системе, поддерживающей многопоточную работу приложений, использование глобальной переменной errno *приведет к тому, что системные вызовы разных потоков будут перезаписывать одну и ту же переменную, то есть ее использование может привести к некорректному результату — на многопоточных системах требуется иная, более сложная, безопасная по отношению к потокам реализация.*

В качестве еще одного дополнительного задания рассмотреть возможные ошибки в имеющихся системных вызовах xv6 *и реализовать получение информации о них. Это потребует глубокого анализа кода ядра* xv6 *и внесения в него соответствующих изменений. Также рекомендуется изучить стандартные коды ошибок в* POSIX *и использовать их там, где это возможно.*

4. *Добавить в* xv6 *системный вызов, возвращающий значение регистра* misa. *Добавить пользовательское приложение, выводящее информацию о подключенных расширениях архитектуры* RISC-V *в соответствии с полученным значением регистра.* Регистр misa является регистром машинного режима и не доступен для чтения из режима супервизора, в т. ч. из ядра ОС xv6. Одним из вариантов реализации является возможность создания и регистрации обработчика программного прерывания или системного вызова машинного режима (чтобы, например, при вызове инструкции ecall из ядра управление передавалось данной функции с повышением привилегий до машинного). Однако такой подход может быть сложным и избыточным, особенно с учетом того, что значение регистра misa не меняется в процессе работы: для простоты его достаточно сохранить в глобальной переменной ядра при загрузке и обращаться к ее значению в системном вызове. Идея состоит в том, чтобы прочитать значение регистра в функции start (файл `start.c`), исполняющейся в машинном режиме, а затем передать в функцию main (файл `main.c`) посредством некоторого свободного регистра общего назначения (например, регистров для передачи аргументов функции или сохраняемых регистров). Для этого следует
   - добавить в файл `riscv.h` функцию чтения значения регистра misa (например, по аналогии с функцией r_mstatus) и функции чтения и записи выбранного пользовательского регистра переда-

чи данных (например, по аналогии с функциями r_tp и w_tp — заметим, что в xv6 регистр tp используется для постоянного хранения идентификатора ядра процессора);

- добавить в функцию start (файл `start.c`) код чтения регистра misa и сохранения значения в выбранном регистре общего назначения;
- добавить в файл main.c глобальную переменную для постоянного хранения значения регистра misa, а в функцию main — сохранение значения выбранного регистра общего назначения в этой переменной так, чтобы код был исполнен только на одном ядре;
- реализовать необходимый системный вызов, возвращающий значение данной глобальной переменной, и пользовательскую утилиту, расшифровывающую значение регистра misa как битовую маску в соответствие с документацией к RISC-V.

Управление процессами

Лабораторная №7: реализация утилиты ps

1. Реализовать системный вызов

```
int ps_count();
```

возвращающий количество запущенных процессов в системе — количество задействованных элементов таблицы процессов (массива структур proc). Для подсчета анализировать состояние процесса: неиспользуемые записи имеют состояние UNUSED. Перед чтением состояния захватывать встроенную в структуру proc блокировку во избежание состояния гонки с другими функциями ядра, которые могут изменять это поле (при создании и завершении процессов, при работе планировщика и др.). Реализовать пользовательскую утилиту, печатающую количество системных вызовов.

Может показаться, что во избежание изменения таблицы процессов (в следствие создания или завершения других процессов) во время ее анализа следовало бы захватывать блокировку всей таблицы. Однако в xv6 *не предусмотрено соответствующей блокировки. Это означает, что такой системный вызов не может дать точное число процессов, работающих в системе, т. е. возвращаемое значение будет оценочным. Однако даже если бы системный вызов давал точное значение на момент запуска или завершения, оно все равно могло бы быть тут же изменено до того, как программа успеет его*

обработать. В то же время длительный захват блокировки всей таблицы процессов может привести к значительным задержкам в работе системы, т. к. в это время будет невозможно создание и завершение процессов. Поэтому такая блокировка с целью достижения точного значения избыточна.

2. Реализовать системный вызов

```
int ps_list (int *pids, int lim);
```

возвращающий массив идентификаторов процессов `pids`, имеющихся в системе. Предельное количество элементов выходного массива задается параметром `lim`. Системный вызов возвращает количество полученных процессов. Если количество процессов окажется больше, чем размер буфера, вернуть отрицательное значение (код ошибки). *Если пользователь не имеет права на запись по указанному адресу буфера, также следует вернуть код ошибки: эти две ситуации разные и должны быть различимы приложением, делающим данный вызов, поэтому следует предусмотреть разные коды ошибок.* Реализация должна учитывать, что при исследовании каждого конкретного процесса — элемента таблицы процессов — следует захватывать встроенную в структуру proc спин-блокировку: захватить блокировку, проверить состояние процесса, если используется — записать его pid, осободить блокировку. *Очень важно освобождать блокировку даже при возникновении ошибки*. Реализовать пользовательскую утилиту, выводящую список идентификаторов процессов, имеющихся в системе. Протестировать ее работу в разных ситуациях (разное количество процессов, недостаточный размер буфера и т. д.).

Общая схема применения данного системного вызова следующая: программа заводит буфер некоторого размера и делает вызов. Если размер буфера недостаточен, нужно завести буфер большего размера и т. д., до тех пор, пока все процессы не будут получены. Можно, конечно, учесть, что ядро xv6 *жестко лимитирует количество процессов в системе, но в решении этой задачи следует избегать использования этого системного параметра, имитируя работу в реальной системе, свободной от такого ограничения.*

Хотя системный вызов из задачи (1) позволяет определить количество процессов на какой-то момент времени, нельзя гарантировать, что их количество не изменится между вызовом ps_count *и* ps_list. *В этом случае могла бы помочь глобальная блокировка таблицы процессов, но ее пришлось бы запрашивать из пользовательского про-*

странства, что чревато большими проблемами: если утилита из-за ошибки или злонамеренно оставит систему в таком состоянии, ее невозможно будет вывести из него, т. к. создание процессов (т. е. запуск команд) и их завершение (в т. ч. принудительное) будет невозможно.

Функционал задачи (1) можно также включить в решение задачи (2): если указатель pids *имеет значение* 0 *(*NULL*) вернуть только количество, но не список процессов. Такой подход характерен для некоторых системных вызовов реальных ОС.*

3. Реализовать системный вызов

```
int ps_info (int pid, struct procinfo *inf);
```

возвращающий информацию о запрошенном процессе: идентификатор процесса, имя процесса, состояние, идентификатор родительского процесса. Разработать соответствующую структуру, доступную из пользовательского пространства и пространства ядра. Системный вызов должен вернуть 0 в случае успеха или отрицательные коды ошибок — отсутствия такого процесса, невозможности записи по указанному адресу. При реализации захватывать необходимые блокировки: блокировку полей процесса персональной блокировкой и глобальную блокировку wait_lock для доступа к родительскому процессу для исключения гонки, связанной с возможной сменой родительского процесса в случае его завершения до анализируемого.

4. Реализовать пользовательскую утилиту ps, печатающую информацию обо всех процессах, запущенных в системе посредством ps_list и ps_info. Протестировать в разных ситуациях, с наличием разных процессов в разных состояниях, в т. ч. в состоянии «зомби».

   Заметим, что между вызовом ps_list *и* ps_info *процессы могут быть как созданы, так и завершены. В первом случае они будут неизбежно проигнорированы утилитой, во втором следует корректно обработать ошибку* ps_info*, чтобы не вывести фиктивные данные.*

5. Реализовать системный вызов

```
int ps_listinfo (struct procinfo *list,
                 int limit);
```

совмещающий функционал задач (1)-(3):

   - если указатель `list` есть 0 (NULL), возвращает число запущенных процессов;

- если указатель `list` есть не 0, записывает в буфер информацию о не более чем `limit` процессах, возвращает их реальное число;
- возвращает код ошибки, если запись по адресу `list` невозможна.

Реализовать утилиту ps посредством данного системного вызова.

Такой системный вызов, с одной стороны, вносит меньше неопределенности в работу утилиты ps*, с другой вносит необходимость одновременного хранения большого объема данных и риски их многократного копирования.*

Функционал задач (3)-(5) можно расширить путем получения дополнительной информации о процессах: открытых файловых дескрипторах, значений сохраненных регистрах пользовательского пространства, контекста процесса (регистрах ядра), а также времени от момента запуска процесса (понадобится расширение структуры proc *для сохранения текущего значения количества тиков в момент вызова* fork*), времени использования процессом процессорного времени (понадобится сохранять значение тиков в момент переключения на процесс с помощью функции* swtch *и инкрементировать счетчик в момент переключения на другой процесс) и другой информацией. Учесть, что ряд полей (например, таблица открытых файловых дескрипторов) не имеет встроенной блокировки, поэтому ее необходимо будет предусмотреть, откорректировав соответствующим образом работу всех системных вызовов, которые могут изменить данную таблицу (*open, close, pipe *и др.).*

В стандарте POSIX *и в ядре* Linux *подобные системные вызовы по историческим причинам не предусмотрены. Вместо этого доступ к списку процессов и их параметрам предоставляется посредством специальной файловой системы —* procfs*, обычно монтируемой в каталог* `/proc`*, где каждому процессу соответствует каталог, имя которого — идентификатор процесса, а файлы и подкаталоги в нем содержат информацию о процессе.*

Управление синхронизацией

Лабораторная №8: реализация мьютексов пользовательского пространства

1. Разработать набор системных вызовов, предоставляющих пользователю функционал синхронизации посредством блокировок с приостановкой, подобных мьютексам.

```
// Создает мьютекс, возвращает его дескриптор
// Возвращает отрицательное число при ошибке
int mutex_create ();
// Блокирует мьютекс по его дескриптору
// Возвращает 0 при успехе, отрицательное число
// при ошибке
int mutex_lock (int md);
// Разблокирует мьютекс по его дескриптору
// Возвращает 0 при успехе, отрицательное число
// при ошибке
int mutex_unlock (int md);
// Освобождает (уничтожает) мьютекс по его
// дескриптору; разблокирует, если он был
// заблокирован. Возвращает 0 при успехе,
// отрицательное число при ошибке
int mutex_close (int md);
```

Для реализации следует создать глобальную таблицу (массив) мьютексов, представляющих собой обертку спящей блокировки и резервировать/освобождать мьютексы в ней. *Использование спин-блокировок в пользовательском пространстве не рекомендуется, т. к. пользовательские процессы могут злоупотреблять длительным их захватом, что приведет к длительному активному ожиданию освобождения с расходом процессорного времени. В то же время пользовательские процессы не имеют ограничения на приостановку, т. к. никогда не исполняются в контексте прерывания.*

В простейшем случае дескриптором мьютекса будет индекс в этой таблице. При этом следует аккуратно блокировать доступ к структуре или ее элементам, не допуская возникновения состояний гонок и взаимоблокировок. В этом случае даже несвязанные иерархией наследования процессы смогут взаимодействовать между собой, зная общий номер мьютекса, но появляется риск злоупотреблений мьютексами при узнавании, угадывании или неудачном совпадении при ошибке программирования номера дескриптора.

Более качественная реализация должна базироваться не только на глобальной таблице мьютексов, но и на таблице используемых (открытых) дескрипторов мьютексов конкретного процесса, ссылающихся на глобальный контейнер мьютексов — для этого следует добавить соответствующую структуру (или воспользоваться принципом «все есть файл» и реализовать функционал мьютексов наравне,

например, с открытыми каналами в пространстве открытых файловых дескрипторов, проведя модификацию соответствующих структур и системных вызовов). В таблице мьютексов следует вести счетчик количества процессов, которые используют данный мьютекс, а также модифицировать вызовы fork и закрытия мьютекса так, чтобы они изменяли значение счетчика соответственно. Вызов exit также следует дополнить автоматическим закрытием всех задействованных дескрипторов мьюеткса процесса. Закрытие мьютекса должно разблокировать его, если он закрывается тем процессом, который его удерживал. Рекомендуется также запретить разблокировать мьютекс, если он заблокирован другим процессом. Если мьютекс был заблокирован на момент вызова fork, считать, что его удерживает родительский процесс. После того, как мьютекс оказывается неиспользуемым, он, как структура ядра, уничтожается (в соответствии со статическим подходом xv6 следует пометить запись в таблице как неиспользуемую). Такой подход позволяет организовать взаимодействие связанных иерархией наследования процессов: процесс создает (открывает) мьютекс, после вызова fork они оказываются доступны по одинаковому номеру дескриптора в родительском и дочернем процессе, каждый из которых должен закрыть его по завершении использования.

Для всех вариантов реализации следует обеспечить инициализацию структур ядра, отвечающих за мьютексы и их блокировки, при старте системы.

Дополнительно, для обеспечения аккуратного взаимодействия не связанных иерархией наследования процессов следует реализовать неанонимные мьютексы — файлы особого рода, которые могут быть созданы и открыты процессами по их имени в VFS. *Для этого следует провести ряд модификаций ядра ОС — добавить соответствующий тип файла, организовать их ассоциацию с глобальной таблицей мьютексов (например, аналогично хранению старшего и младшего номера устройства, ссылки на канал и т. д.), ограничить возможность их «чтения» и «записи» (использования вызовов* read/write*) и т. д.*

Дополнительно можно продумать защиту от взаимоблокировок пользовательских процессов, например, запретить блокировать и разблокировать несколько мьютексов иначе как с помощью одного

вызова, который выстроит их в порядке нумерации в глобальной таблице мьютексов, хотя это может снизить функционал.

2. Написать тестовую программу, которая обменивается строкой между двумя процессами по паре каналов, по одному для каждого направления передачи данных.
   - Родительский процесс должен посимвольно (побайтово) отправить первый параметр командной строки дочернему процессу и закрыть вывод канала.
   - Дочерний процесс должен побайтово считать строку с канала и для каждого байта сразу вывести строку вида

     pid`: received` *char*

     где *pid* — идентификатор его (дочернего) процесса, а *char* — полученный символ; передать байт обратно родительскому процессу в канал и, по достижении конца входного канала, закрыть канал вывода и выйти.
   - Родительский процесс должен получить байты от дочернего, для каждого байта сразу же напечатать «*pid*`: received` *char*» (аналогично дочернему, но со своим *pid*) и выйти.

   Проверить, что без использования синхронизации — блокировки вывода строки на консоль — информация, параллельно выводимая процессами, перемешивается. Реализовать блокировку посредством мьютексов из задачи (1) и убедиться, что информация выводится без перемешивания, каждая строчка отдельная (хотя порядок строк может быть разный от запуска к запуску, это корректно): оба процесса должны захватывать блокировку перед началом вывода каждого сообщения на консоль и освобождать ее сразу после завершения вывода.

   В данной тестовой утилите возможна взаимоблокировка, не связанная с мьютексами: если размер строки превысит размер буфера pipe *в ядре. В реальной ОС можно, например, использовать неблокирующие варианты системного вызова* write *так, чтобы родительский процесс при заполнении канала мог продолжить работу и отложить попытки записи конца сообщения до получения ответов от дочернего процесса. Здесь, в приложении для теста работы мьютексов, достаточно ввести ограничение на длину отправляемого сообщения в соответствии с размером буфера канала в ядре.*

Диагностический буфер сообщений ядра

Лабораторная №9: реализация буфера диагностических сообщений ядра

1. Создать в ядре глобальный строковый буфер для хранения диагностических сообщений (статический, размером кратным размеру физической страницы памяти — точное количество страниц, отводимое на буфер, задавать с помощью макроса в `kernel/param.h`). Буфер должен быть реализован по принципу циклической очереди — предусматривать возможность добавления сообщений (в соответствии со следующими заданиями) так, чтобы при достижении конца (последнего байта) буфера, следующие байты записывались в начало буфера. Таким образом, буфер должен сопровождаться указателем на начало и конец данных в буфере (голову и хвост). Кроме того, буфер должен быть защищен спин-блокировкой для обеспечения очередности вывода сообщений при конкурентом доступе. Реализовать функцию записи байта в буфер. Обеспечить необходимую инициализацию структур буфера при старте системы.
2. Разработать функцию ядра

```
void pr_msg (const char *fmt, ...);
```

добавляющую сообщение в буфер, сопровождая его временем, прошедшим от начала запуска системы (ticks). Функция должна поддерживать те же форматы (спецификаторы), которые поддерживаются функцией ядра printf и автоматически добавлять символ перевода строки (`'\n'`) в конце. Например, при вызове в момент времени 354 и значении переменной `id` 453

```
pr_msg ("id = %d", id);
```

должно вывестись

```
[354] id = 453
```

Учесть, что доступ к ticks следует защищать встроенной в ядро блокировкой.

3. Разработать системный вызов dmesg, копирующий содержимое буфера диагностических сообщений в заданный пользовательский буфер. Системный вызов должен передавать пользователю нультерминированную строку. Реализовать пользовательскую программу dmesg, выводящую содержимое буфера на консоль посредством указанного системного вызова.

4. Протестировать работу утилиты, дополнив какой-либо системный вызов (например, exec) выводом диагностического сообщения о выполняемом им действии в буфер. Убедиться в работоспособности буфера в т. ч. при его заполнении.

В ядре Linux *вывод диагностических сообщений ядра организован именно по такому принципу. Имеется утилита* dmesg*, выводящая содержимое буфера на экран.*

Лабораторная №10: исследование работы ОС с помощью буфера сообщений ядра

1. Добавить в xv6 средства протоколирования событий: прерываний, системных вызовов и переключений контекста процессов. Включение и отключение протоколирования отдельных классов событий должно осуществляться посредством системного вызова и использующей его утилиты пользовательского пространства. Рекомендуется предусмотреть возможность включения протоколирования на определенное время (количество тиков) для экономии места в буфере. Протоколировать следующие классы событий.
   - Системный вызов: сообщается информация о процессе, сделавшем вызов (идентификатор и имя) и номер (или, лучше, имя) системного вызова.
   - Аппаратное прерывание (кроме таймера): сообщается номер прерывания и название устройства, вызвавшего прерывание (UART, virtio), и какое именно событие ввода/вывода вызвало прерывание.
   - Переключение процессов: сообщается на какой процесс (идентификатор и имя) произошло переключение и дамп регистров процесса (trapframe) и ядра процесса (context).
2. Протестировать работу системы с помощью данного средства в различных ситуациях, при наличии различного числа активных процессов, в т. ч. утилиту, создающую взаимодействующие процессы, из лабораторной (8).

   Заметим, что при протоколировании переключения контекста процесса необходимо сохранить захват блокировки основных полей структуры процесса, устанавливаемый ядром. При этом функции протоколирования захватывают глобальный tickslock*. В то же время обработчик прерываний по таймеру — на другом ядре они будут разрешены — может сначала захватить* tickslock*, а затем вызвать*

функцию wakeup *для процессов, остановленных на системном вызове* sleep. *Это создает риск взаимоблокировки при одновременной многопоточности. Поэтому для корректной реализации данной задачи, а также и лабораторной (8), требуется отказ от использования* tickslock *во всем ядре* xv6. *Вместо этого, чтение и изменение глобальной переменной* ticks *следует производить с помощью атомарных инструкций* RISC-V*, реализовав соответствующие ассемблерные функции, или воспользовавшись атомарным расширением компилятора* GCC*, в частности функцией* __sync_fetch_and_add.

Виртуальная память

Лабораторная №11: контроль за доступом к страницам памяти процесса

1. Разработать системный вызов vm_print, печатающий таблицу страниц текущего процесса, с расшифровкой адреса и флагов, в т. ч. флагов прав доступа и наличия доступа к странице. Последний не используется в xv6, необходимо добавить макрос PTE_A в `kernel/riscv.h`.
2. Разработать системный вызов vm_access, который выводит список страниц текущего процесса, к которым был доступ с момента последнего вызова vm_access: при данном системном вызове флаг PTE_A должен сбрасываться. Проверить, что он действительно устанавливается автоматически при работе программы (доступу к переменной, массиву, вызову функции и т. п., в т. ч. для данных, размещенных в стеке, в сегменте данных, в куче).
3. Разработать системные вызовы

```
int vm_get_r (void *ptr);
int vm_get_w (void *ptr);
int vm_get_x (void *ptr);
int vm_get_a (void *ptr);
```

проверяющие установлен ли у физической страницы, на которую отображается виртуальный адрес `ptr`, флаг R (право на чтение), W (право на запись), PTE_X (право на исполнение) и A (был доступ) соответственно. Вызов vm_get_a должен также сбросить флаг A. Если флаг установлен, возвращать `1`, если нет — `0`, если адрес не принадлежит процессу — `-1`. Протестировать работоспособность вызова в различных ситуациях.

4. Реализовать информирование об ошибке доступа к памяти в функции usertrap (`kernel/trap.c`): проанализировать значение регистра scause и, если эта ошибка страницы, вывести информацию о типе ошибки: отсутствие страницы (некорректный виртуальный адрес) или ошибка доступа. Протестировать работу при различных ситуациях.

В качестве дальнейшего развития темы задачи можно реализовать оптимизацию работы системного вызова fork *путем применения технологии копирования при записи (*CoW*) — такая задача предлагается в курсе* MIT. *Для ее решения модифицировать* fork *таким образом, чтобы при создании виртуального адресного пространства дочернего процесса, его страницы создавались в памяти не как копия, а как отображение на одну и ту же физическую страницу. При этом страницы памяти должны быть защищены от записи. При попытке записи и получения соответствующий ловушки следует создать копию страницы в памяти и модифицировать таблицы страниц обоих процессов так, чтобы каждый из них уже мог провести запись в свою копию и продолжить работу (не завершать работу процесса сигналом в этом случае). Предусмотреть, что если страница процесса изначально была защищена от записи, то запись в нее не должна разрешаться этой технологией.*

Ввод и вывод

Лабораторная №12: реализация символьных псевдоустройств

1. Реализовать поддержку ядром xv6 следующих псевдоустройств:
   - null — при чтении всегда возвращает «конец файла», все записываемые данные игнорируются;
   - zero — при чтении возвращает нулевые байты, запись не допускается (возвращается ошибка);
   - uramdom — при чтении возвращает равномерно распределенные случайные байты (можно реализовать датчик случайных чисел по линейному конгруэнтному методу), запись не допускается;
   - dump — при чтении всегда возвращает «конец файла», все записываемые данные передаются в буфер диагностических сообщений ядра (лабораторная 10) в виде шестнадцатеричного побайтового дампа, сопровождаемого текстовым представлением.

Для каждого устройства реализовать драйвер — пару функций для чтения и записи с соответствующим прототипом, зафиксировать старший и младший номера и заполнить таблицу devsw при запуске системы.

Дополнительно рассмотреть возможность реализации использования в xv6 *младших номеров устройств — передачу их в функции чтения и записи с соответствующей модификацией связанных структур и функций ядра. Например, устройства* null, zero *и* dump *могут быть реализованы с одним старшим номером (т. е. в одном драйвере — посредством одной пары функций) с разным поведением в зависимости от младшего номера.*

Также интерес представляет реализация устройства kmsg, *которое при чтении выводит содержимое буфера диагностических сообщений ядра (лабораторная 10), запись не допускается. Для реализации такого устройства нужно позаботиться о том, чтобы для каждого процесса велся счетчик позиции чтения в буфере, а при достижении конца буфера возвращался «конец файла». Для этого также может быть разумно добавить в прототипы функций чтения данных из устройства (и записи данных в устройство) параметр, указывающий позицию.*

2. Обеспечить создание всех устройств при запуске системы (при старте init посредством mknod).
3. Протестировать работоспособность устройств, в т. ч. с использованием утилиты cat.

Лабораторная №13: файловая система — реализация символических ссылок

1. Добавить в xv6 поддержку символических ссылок на файлы. Ограничить длину цели ссылки имеющимся пределом на длину пути — константой MAXPATH. Ограничить максимальное количество переходов по ссылке (глубину рекурсии) некоторым новым параметром (в ядре Linux это ограничение составляет 40 переходов). Добавить новый тип файла — T_SYMLINK в `kernel/stat.h`. Цель ссылки должна храниться в виде обычного содержимого файла типа «ссылка», размер файла определяет длину ссылки (нет необходимости сохранять в файле терминирующий нуль). Разработать следующие системные вызовы по аналогии с работой sys_open, sys_read, sys_write и sys_close:

```
int symlink(const char *target,
            const char *filename);
```

создает символическую ссылку `filename` на `target`, возвращать ошибку при невозможности создать файл ссылки или при превышении максимальной длины цели ссылки.

```
int readlink(const char *filename, char *buf);
```

прочитывает цель символической ссылки в буфер (размер буфера должен быть не меньше, чем MAXPATH). Данный вызов не должен следовать ссылкам рекурсивно и проверять наличие целевого файла, просто прочитать значение цели как есть.

2. Модифицировать системный вызов open таким образом, чтобы по умолчанию он разыменовывал ссылки рекурсивно: при открытии символических ссылок фактически должна открываться цель ссылки. Если цель также ссылка, следовать по ссылкам рекурсивно, при достижении максимального количества переходов по ссылке вернуть ошибку. Если целевой файл не существует, вернуть ошибку.

   При разыменовании учесть, что ссылка может быть относительной — если цель не начинаться с символа `'/'`. В этом случае имя ссылки считается относительно каталога ссылки, а не текущего каталога процесса, разыменовывающего ссылку. Также цель может содержать в пути элементы `".."` (переход на каталог выше) и `"."` (данный каталог).

3. Модифицировать утилиту ln так, чтобы она создавала символические ссылки при указании ключа `-s`:

```
ln -s path target
```

4. Модифицировать утилиту ls так, чтобы она выводила цели символических ссылок и отмечала, если цель не существует. Для обеспечения работы ls необходимо иметь возможность получить информацию (stat) о файле-ссылке, а не только о цели ссылки. Это можно реализовать одним из следующих способов:
   - добавить флаг открытия файла O_NOFOLLOW, при котором open не производит разыменование ссылки, а открывает саму ссылку, после этого можно использовать системный вызов fstat для получения информации о ссылке;
   - добавить системный вызов lstat, который получает информацию о файле по его имени без разыменования ссылок.

5. Добавить приложение для автоматического тестирования символических ссылок: абсолютных и относительных ссылок на файл текущего каталога, нижестоящего каталога, вышестоящего каталога, проверки доступности каждой из них из каталога ссылки, из вышестоящих и нижестоящих каталогов. Протестировать некорректные ссылки: отсутствие целевые файлы, бесконечная рекурсия (ссылка на себя).

Дополнительно можно реализовать поддержку символических ссылок на каталоги, что потребует модификации широкого ряда системных вызовов и утилит.

Лабораторная №14: подсистема блочного ввода/вывода

1. Добавить в xv6 подсчет статистики блочного ввода/вывода: для каждого запущенного процесса добавить счетчик
   - количества прочитываемых с помощью bread блоков — общее количество производимых операций ввода, возможно, выполненных на уровне кеша;
   - количества записываемых с помощью log_write блоков — общее количество производимых операций записи, возможно, выполненных на уровне кеша;
   - количество реально прочитываемых с диска блоков (с помощью virtio_disk_rw);
   - количество реально записываемых на диск блоков (с помощью virtio_disk_rw).
2. Расширить возможности утилиты вывода списка процессов (лабораторная 7) — выводить информацию о количестве прочитанных и записанных процессом блоков (с учетом кешированных операций и без их учета). Протестировать работу в различных ситуациях, подготовив утилиту, производящую значительное количество операций записи и прочтения файла или нескольких, и запустив ее в фоне.

В качестве дополнительного задания на изучение работы файловой системы и блочного вывода/вывода можно добавить в xv6 *поддержку больших файлов, также предлагаемую в курсе* MIT. *Для этого понадобится изменить файловую систему таким образом, чтобы был добавлен еще один уровень косвенности: блок непрямых адресов должен ссылаться не непосредственно на блоки данных, а на еще один блок непрямых адресов, которые уже ссылаются на блоки данных. Кроме того нужно увеличить размер создаваемого образа диска. Это делается настройкой соответствующего параметра ОС (он используется и ядром и утилитой* mkfs). *Определить максимальный под-*

держиваемый при таком подходе размер файла. Реализовать тестовую утилиту, которая будет создавать файл заданного размера с псевдослучайным содержимым, а потом прочитывать и проверять его содержимое путем сверки с той же самой последовательностью псевдослучайных байтов. Проверить возможность чтения и записи файлов с размером, близким к максимальному.

Лабораторная №15: ввод/вывод посредством отображения в память — часы реального времени

Реализовать поддержку часов реального времени в xv6. ***Часы реального времени*** (***RTC***, ***Real-Time Clock***) — аппаратное устройство, отсчитывающее текущее (календарное) время, обычно автономным образом. В виртуальной машине QEMU реализована поддержка виртуальных часов реального времени Goldfish RTC. Показания часов могут быть считаны с помощью отображения регистров устройства в память. Физический адрес регистров устройства начинается с позиции 0x101000. Сами показания часов представляют собой 64-разрядное значение, выражающее количество наносекунд, прошедшее с полуночи 1 января 1970 года. Они записаны в двух регистрах устройства RTC, содержащих, соответственно младшее (LOW) слово и старшее (HIGH) слово (32 бита) значения. Регистр со значением младшего слова расположен непосредственно по адресу 0x101000, старшего — со смещением в 4 байта. За один акт обращения к памяти нужно считывать строго одно значение регистра, при этом сначала считать младшее, а затем старшее слово. Для реализации средств чтения значения RTC следует:

- зарегистрировать константы для позиций регистра младшего и старшего слова значения таймера в файле `memlayout.h`;
- обеспечить отображение регистров RTC в адресное пространство ядра, например, вызвав функцию kvmmap в функции реализации адресного пространства ядра kvmmake (файл vm.c) — по аналогии с тем, как отображаются регистры консоли UART;
- создать в ядре функции чтения регистров младшего и старшего слова значения таймера, например, по аналогии с макросами ReadReg и Reg для консоли UART (файл `uart.c`): важно обратить внимание на используемый тип данных (uint32) и указание ключевого слова volatile, запрещающего использование компилятором оптимизаций исполняемого кода, полагающихся на то, что значение переменной не может быть изменено иначе как самой программой (здесь это условие не выполнено — значение меняется не процессором в результате

исполнения кода программы, а устройством ввода/вывода асинхронно);

- создать в ядре функцию, прочитывающую регистры значения таймера в нужном порядке и формирующую из них корректное 64-разрядное значение времени, а также системный вызов, передающий его в пользовательское пространство;
- добавить утилиту пользовательского пространства, конвертирующее значение времени в человекочитаемую дату (год, месяц, день, час, минута, секунда и доли секунды) и выводящую ее на печать;
- добавить в опции запуска QEMU в `Makefile` ключ `-rtc` со значением, например, `base=localtime` (использование текущего времени хоста в качестве RTC гостевой машины), явным указанием даты запуска и др. вариантами — протестировать работу утилиты для разных значений.

Дополнительно можно реализовать поддержку изменения даты и времени, срабатывания прерывания от таймера в нужный момент времени и других возможностей Goldfish RTC. Для более подробной и точной информацией о поддержке Goldfish RTC следует обращаться к документации и исходному коду QEMU.

Заключение

В данном пособии излагается курс, призванный сформировать первичные представления об устройстве и проектировании операционных систем, а также программировании в пространстве ядра. В каждой лабораторной предложены задания разных уровней сложности — от простых, до достаточно объемных, чтобы рассматривать их как проектные. Помимо предложенных здесь задач рекомендуется также рассмотреть лабораторные, предлагаемые в курсе авторов ОС xv6, доступные на странице MIT. Конкретный набор выполняемых заданий может варьироваться в зависимости от количества часов учебного курса и уровня подготовки целевой аудитории.

Автор выносит благодарность Ловягину Юрию Никитичу за помощь в подготовке текста пособия, а также Национальному Открытому Университету «ИНТУИТ» и Альянсу RISC-V за предоставленную возможность его создания, и анонимным экспертам за ценные советы и замечания по улучшению качества текста и изложения материала.