Содержание

От издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com на странице с описанием соответствующей книги.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу dmkpress@gmail.com. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Apress очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Предисловие

Всем

Добро пожаловать! Надеемся, что вы получите такое же удовольствие от чтения этой книги, какое получили мы, когда ее писали. Книга называется «Операционные системы: три простых элемента» (Operating Systems: Three Easy Pieces) (доступна на сайте http://www.ostep.org), это название, очевидно, является данью уважения одному из самых блестящих в истории курсов лекций (точнее, заметок к ним) – лекциям Ричарда Фейнмана по физике [F96][1]. Без сомнения, этой книге не сравниться с высокой планкой, заданной знаменитым физиком, но, быть может, ее хватит, чтобы удовлетворить взыскующего ответа на вопрос, что такое операционные системы (и системы вообще).

Вот те три простых элемента, которые составляют тематический костяк книги: **виртуализация**, **конкурентность** и **хранение**. Говоря о них, мы в конечном итоге обсудим большую часть того, что делает операционная система; льстим себя надеждой, что попутно вы узнаете много интересного. Ведь узнавать новое всегда интересно, не правда ли? Ну, по крайней мере, должно быть так.

Каждой из основополагающих концепций посвящено несколько глав. В каждой главе представлена одна конкретная проблема и описано ее решение. Главы короткие, и мы старались (в меру возможностей) ссылаться на источники, откуда берут начало идеи. Одна из наших целей при написании этой книги – пролить свет на пути истории, поскольку, на наш взгляд, это поможет студенту лучше понять, что было, что есть и что будет. В данном случае знать, из чего делается колбаса, почти так же важно, как понимать, для чего она предназначена[2].

На протяжении всей книги мы прибегаем к нескольким приемам, о которых, пожалуй, стоит рассказать прямо сейчас. Первый – **существо** проблемы. Прежде чем приступать к решению проблемы, мы всегда пытаемся сформулировать самый важный вопрос; вот это **существо проблемы** явно выделено в тексте, и, хочется надеяться, проблема разрешается с помощью методов, алгоритмов и идей, изложенных далее.

Во многих местах мы объясняем, как система работает, демонстрируя ее поведение во времени. Эта **хронология** очень важна для понимания; уяснив, что происходит, к примеру, при страничном отказе, вы встали на путь понимания принципов работы виртуальной памяти. А осознав, что случается, когда журналируемая файловая система записывает блок на диск, вы сделали первый шаг к овладению таинствами систем хранения.

---

[1] Русский перевод: *Ричард Фейнман.* Дюжина лекций. Шесть попроще и шесть посложней. М.: Лаборатория знаний, 2018.

[2] Подсказка: чтобы ее съесть! Или, если вы вегетарианец, чтобы бежать от нее куда подальше.

В тексте вам также встретится много **отступлений** и **советов** – они расцвечивают магистральную линию изложения. В отступлениях обсуждаются вопросы, имеющие какое-то отношение (пусть и отдаленное) к основному тексту, советы содержат общие уроки, которые могут пригодиться вам при создании собственных систем. Для удобства в конце книги присутствует указатель, в котором сведены все советы и отступления (а заодно и темы, которые мы назвали «существом проблемы»).

В разных местах книги мы используем один из старейших дидактических приемов, **диалог**, как способ представить материал в ином свете. Диалоги позволяют подойти к рассмотрению важных тем (как нам кажется, в увлекательной манере), а иногда резюмировать тот или иной материал. Ну, и дают шанс добавить немного юмора. А уж покажутся они полезными или смешными вам – это совсем другой вопрос.

Каждая часть начинается с описания **абстракции**, предоставляемой операционной системой, а в последующих главах детализируются механизмы, политики и прочие вспомогательные средства, необходимые для реализации этой абстракции. Абстракции – вещь, фундаментальная для всех аспектов информатики, неудивительно, что они играют важнейшую роль и в операционных системах.

Во всех главах мы стараемся приводить **настоящий код** (а не **псевдокод**) всюду, где возможно, так что практически во всех примерах вы сможете набрать код и выполнить его. Исполнение реального кода в реальной системе – лучший способ изучить операционную систему, поэтому мы рекомендуем делать это, если есть возможность. Для лучшего восприятия мы также разместили код по адресу https://github.com/remzi-arpacidusseau/ostep-code.

Мы включили в текст несколько **домашних заданий**, чтобы вы могли оценить, насколько хорошо поняли прочитанное. Часто эти задания представляют собой небольшие **эмуляции** частей операционной системы; скачайте их и выполните для самопроверки. У домашних заданий-эмуляторов есть одна общая черта: задав начальное значение генератора случайных чисел, вы сможете сгенерировать практически бесконечное множество задач; кроме того, эмулятор можно попросить решить задачу за вас. Таким образом, вы можете проверять и перепроверять себя, пока не сочтете, что достигли хорошего уровня понимания.

Самое важное дополнение к этой книге – набор **проектов**, которые на примере проектирования, реализации и тестирования собственного кода позволяют узнать, как работают реальные системы. Все проекты (а равно и вышеупомянутые примеры кода) написаны на **языке программирования C** [KR88]; C – простой и мощный язык, лежащий в основе большинства операционных систем, поэтому его стоит добавить в свой арсенал. Есть два типа проектов (идеи см. в онлайновом приложении). Первый тип – **системное программирование**, эти проекты ориентированы на тех, кто только начинает изучать C и UNIX и хочет узнать о низкоуровневом программировании на C. Второй тип – ядро реальной операционной системы, разработанной в MIT и называемой xv6 [CK+08]; эти проекты ориентированы на студентов, уже знакомых с языком C и желающих покопаться в потрохах ОС. Мы в Висконсинском университете читаем этот курс в трех вариантах:

только системное программирование, только программирование xv6 или то и другое вместе.

Мы потихоньку выкладываем описания проектов и систему тестирования. Подробнее см. на странице https://github.com/remzi-arpacidusseau/ostep-projects. Если вы не студент нашего курса, то эти материалы дадут вам возможность выполнить проекты самостоятельно, чтобы лучше усвоить материал. К сожалению, у вас не будет ассистента кафедры, к которому можно обратиться, если ничего не выходит, но не все в этой жизни можно получить бесплатно (некоторые книги – можно!).

Преподавателям

Если вы преподаватель или профессор, желающий использовать эту книгу, нет никаких препятствий. На всякий случай отметим, что английский текст книги бесплатно размещен на сайте http://www.ostep.org.

Курс довольно хорошо укладывается в 15-недельный семестр, в течение этого времени можно рассмотреть большинство тем на достаточно глубоком уровне. Если вы попытаетесь втиснуть курс в 10-недельную четверть, то, вероятно, какие-то детали из каждой части придется опустить. Есть также несколько глав, посвященных мониторам виртуальных машин, которые мы обычно проходим либо в самом конце большого раздела о виртуализации, либо ближе к концу в качестве отступления.

Трактовка конкурентности в этой книге немного необычна. Во многих книгах по ОС эта тема рассматривается в самом начале, а мы отложили ее до того момента, как студент будет понимать виртуализацию процессора и памяти. Наш опыт чтения этого курса на протяжении почти 20 лет показал, что студентам трудно понять, как возникает проблема конкурентности и почему ее надо решать, пока они не разобрались в том, что такое адресное пространство, что такое процесс и почему контекстное переключение может происходить в любой момент времени. Зато потом понятие потоков и вся возникающая в связи с ними проблематика дается им легко или, по крайней мере, легче.

Во время лекций мы пишем на доске – мелом или маркером. Если лекция посвящена преимущественно теории, то мы приходим на занятия, держа в голове несколько основных идей и примеров, и излагаем их на доске. Конкретные задачи для самостоятельной проработки раздаются студентам в виде заданий. На лекциях более практической направленности мы просто подключаем ноутбук к проектору и показываем реальный код; этот метод особенно хорош при изложении вопросов конкурентности, а также на дискуссионных занятиях, когда студентам демонстрируется код, имеющий отношение к их проектам. Обычно мы не используем слайды, но подготовили набор таковых для тех, кто предпочитает такой стиль изложения.

Если вы захотите получить копию этих материалов, напишите нам по электронной почте. Мы уже разослали их многим преподавателям из разных стран, и кто-то поделился в ответ своими материалами.

И напоследок одна просьба: если вы используете бесплатные онлайновые главы, пожалуйста, давайте только **ссылку** на них, не создавая локальную

копию. Это поможет нам отслеживать, как ими пользуются (за прошедшие несколько лет эти главы скачивали более миллиона раз!), и гарантирует, что студентам будут доступны самые актуальные (и самые лучшие?) версии.

Студентам

Если вы студент, спасибо, что выбрали эту книгу! Для нас большая честь предложить материал, который поможет вам в приобретении знаний об операционных системах. Мы с теплотой вспоминаем об учебниках нашей юности (например, о Hennessy and Patterson [HP90] – классической книге по архитектуре компьютеров) и надеемся, что у вас останутся положительные воспоминания об этой книге.

Вероятно, вы обратили внимание, что англоязычный оригинал данной книги свободно доступен в сети[1]. Тому есть веская причина – учебники очень дороги. Мы надеемся, что эта книга станет первой в череде бесплатных материалов, доступных всем, кто взыскует образования, вне зависимости от того, в какой части мира они проживают и сколько денег готовы потратить на книгу. Ну а если не получится, что ж, и одна бесплатная книга лучше, чем ничего.

Мы также надеемся всюду, где возможно, указывать оригинальные источники представленных в книге материалов: великие статьи и людей, которые на протяжении многих лет формировали научный подход к операционным системам. Идеи не возникают из ничего, это результат труда умных и напряженно работавших людей (включая многих лауреатов премии Тьюринга[2]), поэтому мы должны отдавать должное этим людям и их идеям. Надеемся, что так будет понятнее, какие грандиозные революции имели место, чем если бы мы просто делали вид, что эти идеи были известны всегда [K62]. И быть может, такие отсылки побудят вас к более глубоким самостоятельным изысканиям, ведь чтение основополагающих статей – безусловно, один из лучших способов изучить соответствующую дисциплину.

Глава 1

Диалог о книге

Профессор. *Добро пожаловать! Эта книга называется «Операционные системы: Три простых элемента», а я здесь для того, чтобы научить вас тому, что нужно знать об операционных системах. Меня зовут «Профессор», а тебя?*

Студент. *Добрый день, Профессор! Меня зовут «Студент», как вы, наверное, уже догадались. И я здесь для того, чтобы учиться!*

Профессор. *Звучит неплохо. Есть какие-нибудь вопросы?*

Студент. *А как же! Почему книга называется «Три простых элемента»?*

Профессор. *Это простой вопрос. Видишь ли, существуют знаменитые лекции Ричарда Фейнмана по физике…*

Студент. *А! Тот чувак, что написал «Вы, конечно, шутите, мистер Фейнман», да? Суперская книга! А эта такая же веселая, как та?*

Профессор. *Гм… ну, нет. Та книга великолепна, и я рад, что ты ее прочитал. Льщу себя надеждой, что эта книга больше похожа на его заметки по физике. Некоторые основные положения кратко изложены в книге «Шесть простых элементов». Он писал о физике, а тема этой книги – три простых элемента операционных систем. Это разумно, потому что операционные системы примерно в два раза проще физики.*

Студент. *Что ж, физика мне нравилась, так что это, наверное, неплохо. А что это за элементы?*

Профессор. *Это три ключевые идеи, о которых мы будем говорить:* виртуализация, конкурентность *и* хранение. *По ходу дела мы узнаем, как работает операционная система и, в частности, как она решает, какая программа займет процессор в следующий раз, как система виртуальной памяти помогает справиться с нехваткой памяти физической, как работают мониторы виртуальных машин, как управлять информацией, хранящейся на дисках, и даже немного поговорим о том, как построить распределенную систему, которая продолжает работать, когда некоторые ее части выходят из строя. Такой вот план.*

Студент. *Честно говоря, вообще не понял, о чем вы толкуете.*

Профессор. *И прекрасно! Значит, ты попал точно по адресу.*

Студент. *У меня еще вопрос: как лучше всего научиться всему этому?*

Профессор. *Отличный вопрос! Конечно, у каждого человека свой способ, но вот как поступил бы я сам: записался бы на курс, слушал, как профессор излагает новый материал. Затем в конце каждой недели читал бы эти заметки, чтобы идеи лучше осели в голове. Разумеется, спустя некоторое время (подсказка: до экзамена!) прочитал бы эти заметки снова, чтобы закрепить знания. Без сомнения, профессор будет давать домашние задания и учебные проекты, их надо выполнять; кстати, работа над проектами, в которых нужно писать реальный код для решения реальных задач, – лучший способ воплотить изложенные в этих заметках идеи на практике. Как сказал Конфуций…*

Студент. *А, знаю, знаю! «Я слышу и забываю. Я вижу и запоминаю. Я делаю и понимаю». Или что-то в этом роде.*

Профессор (удивленно). *Откуда ты знаешь, что я собирался сказать?!*

Студент. *Ну, это же напрашивалось. К тому же я большой поклонник Конфуция и еще больший поклонник Сунь Цзы, который, пожалуй, лучше подходит на роль автора этого афоризма*[1].

Профессор (ошеломленно). *Что же, мне кажется, что мы отлично поладим. Просто отлично.*

Студент. *Профессор, еще один вопрос, если позволите. А для чего эти диалоги? В смысле, разве это не просто книга? Почему не излагать материал прямо?*

Профессор. *Ага, превосходный вопрос! Думается мне, что иногда полезно разорвать повествование и немного подумать; эти диалоги как раз на такой случай. Итак, ты и я, вместе, попытаемся разобраться во всех этих довольно сложных идеях. Готов?*

Студент. *Стало быть, нужно будет думать? Что ж, я готов. Да и что мне остается? Не похоже, что эта книга оставит мне много времени на личную жизнь.*

Профессор. *Как и мне, увы. Итак, за работу!*

---

[1] Если верить страничке http://www.barrypopik.com/index.php/new_york_city/entry/tell_me_and_i_forget_teach_me_and_i_may_remember_involve_me_and_i_will_lear/, философ-конфуцианец Сунь Цзы говорил: «Не слышать о чем-то хуже, чем слышать об этом; слышать о чем-то хуже, чем видеть это: видеть что-то хуже, чем знать это; знать что-то хуже, чем воплотить это в действии». Позже это мудрое высказывание почему-то было приписано Конфуцию. Спасибо Сяо Донгу (из Ратгерского университета), рассказавшему нам об этом.

Глава 2

Введение в операционные системы

Коль скоро вы записались на курс по операционным системам, то, надо полагать, уже имеете представление о том, что делает работающая компьютерная программа. Если же нет, то эта книга и соответствующий курс покажутся вам трудными – так что лучше, наверное, будет на время отложить ее, сбегать в ближайший книжный магазин и быстренько проглотить необходимый подготовительный материал (подойдут, например, прекрасные книги Patt & Patel [PP03] и Bryant & O'Hallaron [BOH10]).

Так что же происходит, когда программа запущена?

Работающая программа делает одну очень простую вещь – выполняет команды. Много миллионов (а в наши дни уже миллиардов) раз в секунду процессор **выбирает** команду из памяти, **декодирует** ее (т. е. выясняет, что это за команда) и **выполняет** (т. е. делает то, для чего предназначена команда, например складывает два числа, обращается к памяти, проверяет условие, совершает переход к функции и т. д.). Проделав все это для данной команды, процессор переходит к следующей – и так, пока программа не завершится[1].

Только что мы описали основы модели вычислений **фон Неймана**[2]. Кажется просто, не правда ли? Но на этом курсе мы узнаем, что пока программа работает, происходит много чего еще – и все для того, чтобы системой было **легко пользоваться**.

Существует большой пласт программного обеспечения, которое отвечает за простоту выполнения программ (и даже создание иллюзии, будто одно-

---

[1] Разумеется, современные процессоры совершают много странных и пугающих вещей, чтобы программы работали быстрее, например выполняют несколько команд одновременно и даже, возможно, не по порядку! Но сейчас нас это не интересует; нам важна только простая модель, предполагаемая большинством программ: стороннему наблюдателю представляется, что команды выполняются по одной, последовательно и строго по порядку.

[2] Фон Нейман – один из пионеров создания вычислительных систем. Он также автор пионерских работ по теории игр и атомной бомбе, а еще шесть лет играл в НБА. Ладно, признаемся, что одно из этих утверждений – ложь.

временно работает много программ), за то, чтобы программы могли сообща использовать память, чтобы они могли взаимодействовать с периферийными устройствами и много других вещей. Это программное обеспечение называется **операционной системой** (**ОС**)[1], поскольку несет ответственность за правильное и эффективное функционирование системы, притом так, чтобы использовать ее было легко.

Существо проблемы: как виртуализировать ресурсы

Главный вопрос, на который мы дадим ответ в этой книге, прост: как система виртуализирует ресурсы? Это и есть существо нашей проблемы. *Почему* ОС это делает – не главное, поскольку ответ очевиден: чтобы системой было проще пользоваться. Таким образом, нас будет интересовать, прежде всего, *как*: какие механизмы и политики задействует ОС, чтобы добиться виртуализации? Как ОС достигает эффективности? Какая нужна поддержка со стороны оборудования?

На подобных врезках «существо проблемы» на сером фоне мы будем формулировать конкретные проблемы, решаемые при построении операционной системы. В одной главе может встретиться несколько таких врезок, освещающих проблему с разных сторон. А в тексте главы, конечно, описано решение или, по крайней мере, основные его параметры.

Основной способ, которым ОС решает эту задачу, – общая техника, именуемая **виртуализацией**. Это значит, что ОС берет **физический** ресурс (например, процессор, память или диск) и преобразует его в более общую, более мощную и более простую в использовании **виртуальную форму**. Поэтому иногда операционная система называется **виртуальной машиной**.

Разумеется, чтобы пользователи могли сказать ОС, что делать, и тем самым воспользоваться средствами виртуальной машины (например, выполнить программу, выделить память или обратиться к файлу), ОС должна предоставлять какие-то интерфейсы (API), которые можно вызвать. Типичная ОС экспортирует сотни **системных вызовов**, доступных приложениям. Поскольку ОС предоставляет эти вызовы, чтобы выполнять программы, обращаться к памяти и устройствам и совершать другие подобные действия, иногда говорят, что ОС предоставляет приложениям **стандартную библиотеку**.

Наконец, поскольку виртуализация позволяет запускать много программ (и, стало быть, сообща использовать один центральный процессор), а эти программы могут одновременно обращаться к своим командам и данным (т. е. сообща использовать память), а также к устройствам (т. е. сообща использовать диски и т. д.), то ОС иногда называют **диспетчером ресурсов**. Процессор, память и диск – примеры **ресурсов** системы, а роль операционной системы сводится к тому, чтобы **управлять** этими ресурсами, делая это эффективно или справедливо или стремясь к достижению еще какой-то ведомой ей цели. Чтобы лучше понять роль ОС, рассмотрим несколько примеров.

---

[1] Раньше для ОС были в ходу и другие названия: **супервизор** и даже **главная управляющая программа**. Последнее название звучит уж слишком грозно (посмотрите фильм «Трон», там найдете все детали), так что, к счастью, устоялось название «операционная система».

2.1. Виртуализация процессора

На рис. 2.1 показана наша первая программа. Делает она немного – всего-то вызывает функцию `Spin()`, которая повторно проверяет время и возвращает управление по прошествии одной секунды. Затем программа печатает строку, переданную пользователем в командной строке, и возвращается в начало цикла. И так бесконечно.

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <sys/time.h>
 4 #include <assert.h>
 5 #include “common.h”
 6
 7 int
 8 main(int argc, char *argv[])
 9 {
10     if (argc != 2) {
11         fprintf(stderr, “usage: cpu <string>\n”);
12         exit(1);
13     }
14     char *str = argv[1];
15     while (1) {
16         Spin(1);
17         printf(“%s\n”, str);
18     }
19     return 0;
20 }
```

Рис. 2.1 ❖ Простой пример: программа крутится в цикле и печатает (`cpu.c`)

Допустим, что мы сохранили этот код в файле `cpu.c` и решили откомпилировать и запустить его в системе с одним процессором (иногда мы будем называть его **CPU**). Вот что мы увидим:

```
prompt> gcc -o cpu cpu.c -Wall
prompt> ./cpu “A”
A
A
A
A
^C
prompt>
```

Не особенно интересно – система начинает выполнять программу, которая повторно проверяет время, пока не пройдет одна секунда. Как только секунда прошла, программа печатает строку, переданную пользователем (в данном случае букву «A»), и все повторяется. Эта программа будет работать вечно; лишь нажатием комбинации клавиш **Control-c** (в UNIX-системах

она останавливает программу, работающую в приоритетном режиме) мы можем ее снять.

Теперь повторим эксперимент, но на этот раз запустим несколько экземпляров одной и той же программы. На рис. 2.2 показаны результаты этого чуть более сложного примера.

```
prompt> ./cpu A & ; ./cpu B & ; ./cpu C & ; ./cpu D &
[1] 7353
[2] 7354
[3] 7355
[4] 7356
A
B
D
C
A
B
D
C
A
C
B
D
...
```

Рис. 2.2 ❖ Выполнение сразу нескольких программ

Вот это уже интереснее. Хотя процессор всего один, создается впечатление, что все четыре программы каким-то образом работают одновременно! И как такое чудо могло случиться[1]?

На самом деле операционная система, которой помогает оборудование, создает **иллюзию**, будто в системе имеется очень много виртуальных процессоров. Превращение одного (или нескольких) CPU в кажущееся бесконечным число CPU и как следствие иллюзия одновременного выполнения большого числа программ и называется **виртуализацией процессора**. Это главная тема первого большого раздела книги.

Разумеется, чтобы запускать программы, останавливать их и вообще сообщать ОС, какие программы должны работать, необходимы интерфейсы (API), позволяющие поведать ОС о наших желаниях. Мы будем говорить об этих API на всем протяжении книги, они-то и являются основным способом взаимодействия пользователей с операционной системой.

Возможно, вы заметили, что возможность запускать одновременно несколько программ ставит новые вопросы. Например, если в какой-то момент

---

[1] Обратите внимание, что для запуска четырех процессов одновременно мы воспользовались символом `&`. В оболочке `tcsh` он запускает задание в фоновом режиме, т. е. пользователь может сразу же вводить следующую команду – в нашем случае запустить другую программу. Точка с запятой между командами позволяет запустить несколько программ одновременно. В другой оболочке (например, `bash`) все может быть устроено немного иначе, подробности ищите в документации.

времени хотят работать две программы, какой отдать предпочтение? На этот вопрос отвечает **политика** ОС; политики используются во многих местах для ответа на подобные вопросы, поэтому мы будем изучать их, когда узнаем о базовых **механизмах**, реализуемых операционными системами (в т. ч. механизме, позволяющем выполнять сразу несколько программ). Отсюда и роль ОС как **диспетчера ресурсов**.

2.2. Виртуализация памяти

Теперь переключим внимание на память. Модель **физической памяти** в современных компьютерах очень проста. Память – это просто массив байтов; чтобы **прочитать** из памяти, нужно задать адрес хранящихся в ней данных, а чтобы **записать** в память (или **обновить** ее), нужно дополнительно задать данные, помещаемые по указанному адресу.

Доступ к памяти производится на протяжении всего времени работы программы. Программа хранит все свои данные в памяти и обращается к ним с помощью различных команд: загрузки, сохранения и др. Не забывайте, что все команды программы тоже хранятся в памяти, поэтому доступ к памяти производится при выборке каждой команды.

Рассмотрим программу (рис. 2.3), которая выделяет память путем обращения к функции `malloc()`. Вот что она выводит на экран:

```
prompt> ./mem
(2134) адрес, на который указывает p: 0x200000
(2134) p: 1
(2134) p: 2
(2134) p: 3
(2134) p: 4
(2134) p: 5
^C
```

Эта программа делает две вещи. Сначала она выделяет блок памяти (строка a1). Затем печатается адрес этого блока (a2), после чего в первый элемент только что выделенного блока записывается нуль (a3). Наконец, программа возвращается в начало цикла, ожидает в течение одной секунды и увеличивает на единицу значение, хранящееся по адресу, который находится в `p`. В каждом предложении печати выводится также идентификатор процесса (PID) выполняемой программы. У каждого работающего процесса имеется свой уникальный PID.

И снова результат не особенно впечатляет. Блок памяти выделен по адресу `0x200000`. По ходу работы программа медленно обновляет значение и печатает результат.

Теперь, как и раньше, запустим несколько экземпляров этой программы и посмотрим, что будет (рис. 2.4). Из распечатки видно, что обе работающие программы выделили блок памяти по одному и тому же адресу (`0x200000`), но каждая обновляет значение по адресу `0x200000` независимо! Складывается

впечатление, что работающие программы не разделяют одну и ту же физическую память, а владеют собственной частной памятью[1].

```
 1 #include <unistd.h>
 2 #include <stdio.h>
 3 #include <stdlib.h>
 4 #include “common.h”
 5
 6 int
 7 main(int argc, char *argv[])
 8 {
 9     int *p = malloc(sizeof(int)); // a1
10     assert(p != NULL);
11     printf(“(%d) адрес, на который указывает p: %p\n”,
12            getpid(), p); // a2
13     *p = 0; // a3
14     while (1) {
15         Spin(1);
16         *p = *p + 1;
17         printf(“(%d) p: %d\n”, getpid(), *p); // a4
18     }
19     return 0;
20 }
```

Рис. 2.3 ❖ Программа, обращающаяся к памяти

```
prompt> ./mem &; ./mem &
[1] 24113
[2] 24114
(24113) адрес, на который указывает p: 0x200000
(24114) адрес, на который указывает p: 0x200000
(24113) p: 1
(24114) p: 1
(24114) p: 2
(24113) p: 2
(24113) p: 3
(24114) p: 3
(24113) p: 4
(24114) p: 4
...
```

Рис. 2.4 ❖ Выполнение нескольких экземпляров программы обращения к памяти

И действительно, именно это здесь и происходит – вследствие **виртуализации памяти**, осуществляемой ОС. Каждый процесс обращается к собст-

[1] Чтобы этот пример работал, как показано, необходимо отключить рандомизацию адресного пространства. Рандомизация – это механизм защиты от некоторых видов атак. Почитайте об этом самостоятельно, особенно если хотите узнать, как можно взломать компьютерную систему путем атаки с переполнением стека. Не подумайте, что это рекомендация...

венному частному **виртуальному адресному пространству** (иногда оно называется просто **адресным пространством**), которое ОС каким-то образом отображает на физическую память машины. Обращение к памяти внутри одной программы никак не влияет на адресное пространство других процессов (и самой ОС); с точки зрения работающей программы, вся физическая память принадлежит ей. Но на самом деле физическая память – это разделяемый ресурс, управляемый операционной системой. Как именно она это делает, также обсуждается в первой части книги, посвященной **виртуализации**.

2.3. Конкурентность

Вторая из основных тем этой книги – **конкурентность**. Мы пользуемся этим общим термином для обозначения совокупности проблем, которые возникают и должны решаться, когда одновременно (т. е. конкурентно) в одной программе производятся действия с несколькими объектами. Первоначально проблемы конкурентности возникли в самой операционной системе; в приведенных выше примерах мы видели, что ОС манипулирует сразу несколькими сущностями: запускает один процесс, потом другой и т. д. Как выясняется, это поднимает ряд глубоких и интересных вопросов.

К сожалению, проблемы конкурентности вышли за пределы самой операционной системы. И в современных **многопоточных** программах возникают те же самые проблемы. Продемонстрируем их на примере программы на рис. 2.5.

Хотя в данный момент вы, возможно, не понимаете все детали этого кода (мы разберемся с ними в разделе, посвященном конкурентности), основная идея проста. Главная программа создает два **потока**, вызывая функцию `Pthread_create()`[1]. Потоки можно представлять себе как функции, одновременно работающие в одном и том же пространстве памяти. В примере выше каждый поток запускает функцию `worker()`, которая просто `loops` раз увеличивает счетчик в цикле.

Существо проблемы: как правильно писать конкурентные программы

Как правильно написать программу, если в одном и том же пространстве памяти может конкурентно выполняться несколько потоков? Какие примитивы ОС нам для этого понадобятся? Какие механизмы должно обеспечить оборудование? Как ими воспользоваться для решения проблем конкурентности?

Ниже показано, что печатает эта программа, когда переменной `loops` присвоено начальное значение 1000. Значение `loops` определяет, сколько раз

[1] Вообще-то, функция правильно называется `pthread_create()`; имя, начинающееся заглавной буквой, принадлежит нашей обертке, которая вызывает `pthread_create()` и проверяет, что та завершилась успешно. Детали см. в коде.

каждый из исполнителей (экземпляров функции `worker`) увеличивает разделяемый счетчик. Если вначале оно равно 1000, то как вы думаете, каким будет конечное значение `counter`?

```
prompt> gcc -o thread thread.c -Wall -pthread
prompt> ./thread 1000
Начальное значение : 0
Конечное значение  : 2000
```

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include “common.h”
 4
 5 volatile int counter = 0;
 6 int loops;
 7
 8 void *worker(void *arg) {
 9     int i;
10     for (i = 0; i < loops; i++) {
11         counter++;
12     }
13     return NULL;
14 }
15
16 int
17 main(int argc, char *argv[])
18 {
19     if (argc != 2) {
20         fprintf(stderr, “usage: threads <value>\n”);
21         exit(1);
22     }
23     loops = atoi(argv[1]);
24     pthread_t p1, p2;
25     printf(“Начальное значение : %d\n”, counter);
26
27     Pthread_create(&p1, NULL, worker, NULL);
28     Pthread_create(&p2, NULL, worker, NULL);
29     Pthread_join(p1, NULL);
30     Pthread_join(p2, NULL);
31     printf(“Конечное значение : %d\n”, counter);
32     return 0;
33 }
```

Рис. 2.5 ❖ Многопоточная программа (`threads.c`)

Вы, наверное, уже догадались, что по завершении обоих потоков конечное значение `counter` будет равно 2000, поскольку каждый поток увеличил счетчик 1000 раз. Действительно, если начальное значение `loops` равно N, то следует ожидать, что в конце программы счетчик окажется равным $2N$. Но, как выясняется, жизнь устроена сложнее. Давайте-ка запустим ту же программу с бóльшим значением `loops` и посмотрим, что произойдет:

```
prompt> ./thread 100000
Начальное значение : 0
Конечное значение  : 143012 // странно
prompt> ./thread 100000
Начальное значение : 0
Конечное значение  : 137298 // что за черт?!
```

На этот раз мы задали начальное значение 100 000, но вместо конечного значения 200 000 получили 143 012. А запустив программу еще раз, мы мало того что получили *неверное* значение, так еще и *отличающееся* от полученного в первый раз. На самом деле если запускать программу снова и снова с большим значением `loops`, то иногда будет даже получаться правильный ответ! Так что же происходит?

Причина таких странных результатов связана с тем, что команды выполняются по одной. Но, к сожалению, самая главная часть этой программы – та, где счетчик увеличивается на единицу, – требует трех команд: первая загружает значение счетчика из памяти в регистр, вторая увеличивает его, а третья записывает обратно в память. Поскольку эти три команды выполняются не **атомарно** (как неделимое целое), происходят странные вещи. Именно эту проблему конкурентности мы будем подробнейшим образом рассматривать во второй части книги.

2.4. Хранение

Третья из главных тем этого курса – **хранение**. В памяти системы данные легко могут быть утрачены, потому что такие запоминающие устройства, как ДЗУПВ (англ. *DRAM*), являются **энергозависимыми** – если питание пропадает или система внезапно выходит из строя, то все данные в памяти теряются. Таким образом, нам нужно, чтобы оборудование и программное обеспечение могли сохранять данные на постоянной основе; подобные системы хранения – важнейшая часть любой системы, т. к. пользователи дорожат своими данными.

Оборудование выступает в форме того или иного устройства **ввода-вывода**; в современных системах типичным хранилищем долговечной информации является **жесткий диск**, хотя постепенно на эту роль выдвигаются **твердотельные запоминающие устройства** (англ. *SSD*).

Та часть операционной системы, которая занимается управлением диском, называется **файловой системой**, она отвечает за надежное и эффективное хранение созданных пользователями **файлов** на системных дисках.

В отличие от абстракций процессора и памяти, ОС не создает частного виртуализированного диска для каждого приложения. Напротив, предполагается, что пользователи зачастую желают **разделять** информацию, находящуюся в файлах, т. е. пользоваться ей совместно. Например, при написании программы на C вы сначала используете редактор (например, Emacs[1]), чтобы

[1] Верим, что вы используете именно Emacs. Если вы пользуетесь vi, то, видимо, с вами что-то не так. А уж если вы работаете с чем-то, вовсе не являющимся настоящим редактором кода, то дела обстоят совсем плохо.

создать и редактировать C-файл (`emacs -nw main.c`). Затем вы запускаете компилятор, чтобы преобразовать исходный код в исполняемый (например, `gcc -o main main.c`). Покончив с этим, вы запускаете новый исполняемый файл (например, `./main`). Таким образом, одни и те же файлы используются разными процессами. Сначала Emacs создает файл, который подается на вход компилятору, затем компилятор создает из него новый исполняемый файл (для чего требуется много шагов, о которых вы можете узнать на курсе по компиляторам), и, наконец, этот исполняемый файл запускается. Так рождается новая программа!

Чтобы лучше разобраться в этом, возьмем какой-нибудь код. На рис. 2.6 показана программа создания файла (`/tmp/file`), содержащего строку «hello world».

```
 1 #include <stdio.h>
 2 #include <unistd.h>
 3 #include <assert.h>
 4 #include <fcntl.h>
 5 #include <sys/types.h>
 6
 7 int
 8 main(int argc, char *argv[])
 9 {
10     int fd = open("/tmp/file", O_WRONLY | O_CREAT | O_TRUNC, S_IRWXU);
11     assert(fd > -1);
12     int rc = write(fd, "hello world\n", 13);
13     assert(rc == 13);
14     close(fd);
15     return 0;
16 }
```

Рис. 2.6 ❖ Программа ввода-вывода (`io.c`)

Существо проблемы: как сохранять данные на постоянной основе

Файловая система – это часть ОС, отвечающая за управление постоянно хранимыми данными. Какие методы необходимы для корректного решения этой задачи? Какие механизмы и политики позволят добиться при этом высокой производительности? Как обеспечить надежность в условиях возможных отказов оборудования и программ?

Для решения этой задачи программа трижды обращается к операционной системе. Первый вызов, `open()`, создает файл и открывает его, второй, `write()`, записывает в файл данные, а третий, `close()`, закрывает файл и тем самым дает программе знать, что больше операций записи в него не будет. Эти **системные вызовы** передаются части ОС, которая называется **файловой системой**, она обрабатывает запросы и возвращает программе код успеха или ошибки.

Вам, наверное, любопытно, что именно делает ОС, для того чтобы записать данные на диск. Мы покажем, но обещайте сначала закрыть глаза – непри-

ятное это зрелище. Файловая система должна проделать немало работы: сначала она вычисляет, в каком месте диска должны оказаться новые данные, а затем работает с различными структурами данных. Для этого необходимо отправлять команды ввода-вывода физическому запоминающему устройству, чтобы либо прочитать существующие структуры, либо обновить их. Всякий, кому доводилось писать **драйвер устройства**[1], знает, что заставить устройство что-то сделать от имени пользователя – сложная процедура, изобилующая тонкими деталями. Требуется глубокое знание низкоуровневого интерфейса устройства и его точной семантики. По счастью, ОС предлагает стандартный и простой способ доступа к устройствам с помощью системных вызовов. Поэтому ОС иногда рассматривают как **стандартную библиотеку**.

Конечно, поверх собственно устройств файловая система надстраивает много механизмов доступа к устройствам и управления данными. Из соображений производительности большинство файловых систем на некоторое время откладывают операции записи в надежде сформировать из них более крупные группы. Чтобы справиться с крахом системы во время записи, многие файловые системы включают хитроумные протоколы записи, например **журналирование** или **копирование при записи**, тщательно упорядочивая операции записи на диск таким образом, чтобы в случае сбоя система могла гарантированно восстановиться в каком-то разумном состоянии. Для повышения эффективности типичных операций файловые системы прибегают к разнообразным структурам данных и методам доступа, от простых списков до сложных B-деревьев. Если пока это вам ни о чем не говорит, не страшно! Мы разберемся во всем этом гораздо подробнее в третьей части книги, где будем обсуждать устройства и ввод-вывод сначала в общем, а затем на примере конкретных дисков, RAID-массивов и файловых систем.

2.5. Цели проектирования

Итак, теперь у вас есть представление о том, что же делает ОС: она берет физические **ресурсы** (процессор, память или диск) и **виртуализирует** их. Она берет на себя сложные вопросы, связанные с **конкурентностью**. И **хранит** файлы, обеспечивая безопасность данных на долгое время. Поскольку мы хотим построить такую систему, необходимо сформулировать цели, на достижении которых сосредоточить усилия по проектированию и реализации, идя по мере необходимости на компромиссы. Отыскание подходящего набора компромиссов – ключ к проектированию систем.

Одна из самых важных целей – выбор абстракций, при которых системой будет легко и удобно пользоваться. Абстракции – вообще основа всего в информатике. Благодаря абстрагированию мы можем написать большую программу, разделив ее на меньшие части, которые проще понять. Благо-

[1] Драйвером устройства называется часть операционной системы, которая знает, как обращаться с конкретным устройством. Ниже мы еще поговорим об устройствах и их драйверах.

даря ему же мы можем писать программы на языках высокого уровня типа C[1], не думая о языке ассемблера, писать код на языке ассемблера, не думая о логических вентилях, и собирать процессор из вентилей, не «заморачиваясь» мыслями о транзисторах. Абстрагирование – настолько фундаментальная вещь, что иногда забывают о его важности, но мы такого не допустим и в каждом разделе будем обсуждать, какие абстракции уже разработали. Это поможет вам расуждать о частях ОС.

Одна из целей проектирования и реализации операционной системы – обеспечить высокую **производительность**, или, как еще говорят, **минимизировать накладные расходы** ОС. Виртуализация и удобство пользования – достойные цели, но не любой ценой; необходимо, чтобы эти и другие механизмы ОС не сопровождались непомерными накладными расходами. Накладные расходы могут принимать разную форму: перерасход времени (больше команд) или перерасход места (в памяти или на диске). Мы будем искать решения, которые минимизируют одно, другое или то и другое сразу, если возможно. Однако совершенство не всегда достижимо, мы научимся подмечать это и (там, где это приемлемо) мириться с этим.

Еще одна цель – обеспечить **защиту** приложений друг от друга и ОС от приложений. Поскольку мы хотим, чтобы много программ могли работать одновременно, требуется гарантировать, что намеренное или случайное некорректное поведение одной программы не нанесет урона другим; и уж точно мы не хотим, чтобы какое-то приложение могло повредить саму ОС (поскольку это отразилось бы на *всех* вообще программах, работающих в системе). Защита – сердцевина одного из главных принципов операционной системы – **изоляции**; изолирование процессов друг от друга – ключевая характеристика защиты и потому определяет многое из того, что должна делать ОС.

Операционная система должна работать бесперебойно; если она отказывает, то отказывают сразу *все* работающие в системе приложения. Поэтому ОС стремятся обеспечить высочайший уровень **надежности**. Поскольку со временем операционные системы только усложняются (иногда они содержат миллионы строк кода), построение надежной ОС – очень трудная задача, огромное число исследований в этой области (в т. ч. и наши собственные работы [BS+09, SS+10]) посвящены именно этой проблеме.

Есть и другие цели: **энергоэффективность**, которая приобретает особую важность в мире, который все сильнее озабочен экологией; **безопасность** (в действительности обобщение защиты) в условиях, когда существуют вредоносные приложения, особенно в наши времена повсеместного распространения сетей; **мобильность** становится все важнее, поскольку ОС работают на устройствах все меньшего и меньшего размера. В зависимости от характера использования системы стоящие перед ОС цели разнятся и потому реализуются несколько различающимися способами. Однако, как мы увидим, многие описанные нами принципы построения ОС применимы к широкому кругу различных устройств.

---

[1] Возможно, кто-то не согласится с тем, что C – язык высокого уровня. Но напомним, что это курс по ОС, так что мы должны быть счастливы уже тем, что не приходится всю дорогу писать на ассемблере!

2.6. Немного истории

Прежде чем расстаться с этим введением, мы хотели бы изложить краткую историю развития операционных систем. Как и для любой созданной людьми системы, хорошие идеи аккумулировались на протяжении времени, по мере того как инженеры осознавали, что именно важно при проектировании ОС. Здесь мы обсудим лишь несколько наиболее заметных вех. Более полный рассказ можно найти в великолепной истории операционных систем, написанной Бринчем Хансеном [BH00].

Первые операционные системы: просто библиотеки

В самом начале операционные системы делали не слишком много. По существу, это был просто набор библиотек, содержащий часто используемые функции. Например, чтобы не заставлять каждого программиста писать низкоуровневый код ввода-вывода, «ОС» предоставляла соответствующие API, облегчая тем самым жизнь разработчикам.

Обычно на этих старых больших машинах – мейнфреймах – в каждый момент времени работала одна программа, и управлял всем этим оператор-человек. Многое из того, что делала бы современная ОС (например, решение о том, в каком порядке запускать задания), возлагалось на оператора. Разумный разработчик завязал бы хорошие отношения с оператором, чтобы тот передвинул его задание в начало очереди.

Такой режим обработки назывался **пакетным**, поскольку задания собирались оператором в «пакет», который потом запускался на выполнение. В то время еще не было интерактивного режима работы, потому что заставлять пользователя сидеть перед компьютером и время от времени взаимодействовать с ним было бы попросту слишком дорого – ведь большую часть времени он бы ничего не делал, а вычислительному центру это обходилось бы в сотни долларов в час [BH00].

Не только библиотеки: защита

Перерастая простую библиотеку общеупотребительных служб, операционные системы начать играть более заметную роль в управлении машинами. И тут важно отметить осознание того, что код, работающий от имени ОС, имел особенности; поскольку он управлял устройствами, с ним следовало обращаться не как с кодом обычного приложения. Почему? Представьте, что любому приложению разрешено читать любой участок диска; тогда ни о какой конфиденциальности не может быть и речи, ведь любая программа может прочитать любой файл. Поэтому реализация **файловой системы** (с целью управления вашими файлами) в виде библиотеки не имеет смысла. Нужно было что-то другое.

Так возникла идея **системного вызова**, которая впервые была опробована в вычислительной системе Atlas [K+61, L78]. Вместо того чтобы реализовывать подпрограммы ОС в виде библиотеки (и для доступа к ним выполнять просто **вызов процедуры**), было предложено включить две специальные аппаратные команды и аппаратное же состояние, позволяющие переходить в ОС более формальным и контролируемым способом.

Ключевое отличие системного вызова от вызова процедур заключается в том, что системный вызов передает управление (т. е. совершает переход) ОС и одновременно увеличивает **аппаратный уровень привилегий**. Пользовательские приложения работают в так называемом **режиме пользователя**, когда оборудование ограничивает доступные приложению возможности; например, приложение, работающее в режиме пользователя, обычно не может инициировать запрос ввода-вывода к диску, обратиться к произвольной странице физической памяти или отправить пакет в сеть. Когда произведен системный вызов (обычно с помощью специальной команды **системного прерывания** (англ. *trap*), оборудование передает управление предопределенному **обработчику системных прерываний** (который ОС предварительно подготовила) и в то же время поднимает уровень привилегий до **режима ядра**. В режиме ядра ОС имеет полный доступ ко всему оборудованию и, следовательно, может инициировать запрос ввода-вывода, выделить программе дополнительную память и т. д. Закончив обслуживать запрос, ОС возвращает управление пользователю с помощью специальной команды **возврата из системного прерывания**, которая не только возобновляет работу с того места, где приложение прервалось, но и восстанавливает режим пользователя.

Эра мультипрограммирования

Настоящий взлет операционные системы пережили, когда на смену мейнфреймам пришли **мини-компьютеры**. Классические машины, в частности семейство PDP компании Digital Equipment, сделали компьютеры гораздо более доступными по цене; теперь вместо одного мейнфрейма на всю крупную организацию стало возможно выделить отдельный компьютер сравнительно небольшому коллективу. Неудивительно, что одним из главных последствий снижения стоимости стало возрастание активности разработчиков – у большего числа талантливых людей появилась возможность дорваться до компьютера, и в результате вычислительные системы стали делать более интересные и красивые вещи.

В частности, широкое распространение получило **мультипрограммирование** – вследствие желания более эффективно использовать машинные ресурсы. Вместо того чтобы выполнять задания по одному, ОС научились загружать несколько заданий в память и быстро переключаться между ними, повышая тем самым коэффициент использования процессора. Это переключение было особенно важно, потому что драйверы ввода-вывода работали медленно, а заставлять программу ждать, пока будет обслужено устройство ввода-вывода, значило бы непроизводительно расходовать процессорное

время. Почему бы вместо этого не переключиться на другое задание и не дать ему поработать?

Стремление поддержать мультипрограммирование и перекрытие во времени при наличии ввода-вывода и прерываний повлекло за собой концептуальное развитие операционных систем сразу в нескольких направлениях. Приобрели важность такие вопросы, как **защита памяти**; мы же не хотим, чтобы одна программа могла обращаться к памяти другой. Не менее важно было понять, как решать проблемы **конкурентности**, сопутствующие мультипрограммированию; во весь рост встал вопрос о том, как заставить ОС корректно вести себя, несмотря на прерывания. Мы изучим эти и смежные вопросы далее в этой книге.

Одним из главных практических достижений в то время стало появление операционной системы UNIX в основном благодаря усилиям Кена Томпсона (и Денниса Ритчи) из компании Bell Labs (да-да, телефонной компании). UNIX заимствовала много хороших идей у других операционных систем (в особенности Multics [O72] и кое-что у системы TENEX [B+72] и Berkeley Time-Sharing System [S+68]), но упростила их и сделала использование более удобным. Вскоре эта команда начала рассылать магнитные ленты с исходным кодом UNIX людям по всему миру, многие из них затем подключились к работе и начали вносить в систему свои добавления; подробнее см. врезку «Отступление» ниже[1].

Современность

На мини-компьютерах история не закончилась, вскоре появились машины нового типа – дешевле, быстрее и рассчитанные на массовое производство: **персональные компьютеры**, или **ПК**, как мы их сегодня называем. Вслед за ранними машинами Apple (например, Apple II) и IBM PC эта новая порода машин очень быстро завоевала доминирующие позиции на рынке вычислительной техники, поскольку низкая стоимость позволяла иметь отдельную машину на каждом рабочем столе вместо общего мини-компьютера на всю группу.

К сожалению, с точки зрения операционных систем, ПК поначалу знаменовали гигантский отскок назад, поскольку первые системы забыли (или никогда не знали) уроки, выученные в эру господства мини-компьютеров. Например, ранние операционные системы, в частности **DOS** (**Disk Operating System** от компании **Microsoft**), не задумывались о важности защиты памяти, а потому злонамеренное (или просто плохо написанное) приложение могло затереть всю память. В первых поколениях **Mac OS** (v9 и более ранних) был принят кооперативный подход к планированию заданий, поэтому поток, который случайно зацикливался, мог остановить всю систему, так что ее приходилось перезагружать. Скорбный список функций ОС, отсутствующих в этом поколении систем, длинен, слишком длинен, чтобы обсуждать его здесь подробно.

---

[1] Мы используем врезки «Отступление» и другие, чтобы привлечь внимание к различным фактам, которые не укладываются в основной поток изложения. Иногда они просто дают возможность пошутить, ведь нет же ничего плохого в том, чтобы немного рассеяться? Да, наверное, многие шутки неудачны.

К счастью, после нескольких лет страданий давно известные механизмы операционных систем для мини-компьютеров начали постепенно проникать и на рабочий стол. Например, в основу системы Mac OS X/macOS была положена UNIX со всей функциональностью, которую можно ожидать от такой зрелой системы. Windows также восприняла многие из величайших идей в истории вычислительной техники, и началось это с Windows NT, знаменовавшей гигантский скачок к технологии ОС от Microsoft. Даже современные сотовые телефоны управляются операционными системами (например, Linux), которые гораздо больше напоминают мини-компьютеры 1970-х годов, чем ПК 1980-х (и слава богу); приятно видеть, что хорошие идеи, заложенные в лучшую пору разработки ОС, проложили путь в современный мир. А еще приятнее, что эти идеи продолжают развиваться, обретают новые возможности и делают современные системы еще удобнее для пользователей и приложений.

Отступление: важность Unix

Невозможно переоценить важность Unix в истории операционных систем. Испытав влияние более ранних систем (в частности, знаменитой системы **Multics**, созданной в MIT), Unix объединила в себе многие великие идеи, что позволило создать систему одновременно простую и мощную.

В основу оригинальной Unix от «Bell Labs» лег универсальный принцип, заключающийся в построении небольших узкоспециализированных программ, которые можно было соединять для решения более крупных задач. **Оболочка**, в которой вводятся команды, предоставляла такие примитивы, как **конвейеры**, открывавшие возможность для подобного программирования на метауровне. Например, чтобы найти все строки текстового файла, в которых встречается слово «foo», а затем подсчитать количество таких строк, нужно было ввести команду `grep foo file.txt|wc --l`, в составе которой используются программы `grep` и `wc` (word count – счетчик слов).

Среда Unix была удобна для программистов и разработчиков и предлагала также компилятор нового **языка программирования C**. Благодаря простоте написания программ и обмена ими Unix приобрела невероятную популярность. И наверное, тому сильно способствовала готовность авторов бесплатно раздавать копии всем, кто попросит, – ранняя форма **ПО с открытым исходным кодом**.

Также очень важную роль сыграли доступность и удобочитаемость кода. Элегантное небольшое ядро, написанное на C, как бы приглашало всех желающих к экспериментам, к добавлению новой «крутой» функциональности. Например, инициативная группа в Беркли под руководством **Билла Джоя** создала потрясающий дистрибутив (**Berkeley Systems Distribution**, или **BSD**), в который вошла передовая система виртуальной памяти, файловая система и сетевая подсистема. Впоследствии Джой стал одним из основателей компании **Sun Microsystems**.

К сожалению, распространение Unix немного замедлилось, поскольку компании пытались закрепить за собой права собственности и извлекать прибыль – печальное (но типичное) следствие привлечения юристов. Многие компании начали выпускать собственные варианты: **SunOS** от Sun Microsystems, **AIX** от IBM, **HPUX** (или «H-Pucks») от HP, **IRIX** от SGI. Из-за судебных тяжб между AT&T/Bell Labs и другими игроками над Unix нависли тучи, и многие сомневались, выживет ли она, особенно в связи с появлением Windows, воцарившейся на рынке ПК.

Отступление: и тогда пришла LINUX

К счастью для Unix, молодой финский хакер **Линус Торвальдс** решил написать собственную версию Unix. Он позаимствовал у оригинальной системы многие принципы и идеи, но не кодовую базу и тем самым избежал юридических сложностей. Он рекрутировал в добровольные помощники многих программистов со всего мира, воспользовался развитыми инструментами GNU, уже существовавшими к тому времени [G85], и вскоре родилась ОС **Linux** (а вместе с ней современное движение за ПО с открытым исходным кодом).

Когда настала эра интернета, многие компании (например, Google, Amazon, Facebook и другие) выбрали Linux, поскольку она распространялась свободно и ее можно было модифицировать под свои нужды; трудно представить, что все эти компании добились бы такого успеха, если бы подобной системы не существовало. Когда смартфоны стали доминирующей платформой, ориентированной на пользователя, Linux закрепилась и там тоже (посредством Android) – и по тем же причинам. А Стив Джобс забрал свою основанную на Unix операционную среду **NeXTStep** в Apple и тем самым сделал Unix популярной на настольных компьютерах (хотя многие пользователи технологий Apple, возможно, и не подозревают об этом). Таким образом, Unix продолжает здравствовать и даже более важна, чем когда-либо прежде. Компьютерные боги, если вы в них верите, должны быть благодарны за это великолепное подношение.

2.7. Резюме

Итак, краткое введение в ОС позади. Благодаря современным операционным системам с компьютерами стало относительно легко работать, и практически все используемые сегодня операционные системы так или иначе испытали влияние идей, которые мы будем обсуждать далее в этой книге.

К сожалению, из-за ограничений по времени мы не сможем рассмотреть все части ОС. Например, в операционной системе много **сетевого** кода, но чтобы подробнее узнать о нем, вам придется записаться на курс по сетям. Не менее важны **графические** устройства, походите на курс по графике, чтобы расширить свои знания в этом направлении. Наконец, в некоторых книгах по операционным системам большое внимание уделено **безопасности**; мы тоже затронем эту тему, поскольку ОС должна защищать работающие программы друг от друга и давать пользователям возможность защитить свои файлы, но не станем глубоко вдаваться в детали, которые излагаются в курсе по безопасности.

Однако же все равно остается много важных тем для изучения, включая основы виртуализации процессора и памяти, конкурентность и постоянное хранение на запоминающих устройствах с помощью файловых систем. Не пугайтесь! Да, нам предстоит длинная дорога, но это будет интересное путешествие, и в конце пути вы сможете по-новому взглянуть на то, как в действительности работают вычислительные системы. А теперь – за работу!

Литература

[BS+09] «Tolerating File-System Mistakes with EnvyFS» by L. Bairavasundaram, S. Sundararaman, A. Arpaci-Dusseau, R. Arpaci-Dusseau. USENIX '09, San Diego, CA, June 2009. *Интересная статья о том, как использовать сразу несколько файловых систем на случай, если в одной произойдет ошибка.*

[BH00] «The Evolution of Operating Systems» by P. Brinch Hansen. В книге «Classic Operating Systems: From Batch Processing to Distributed Systems». Springer-Verlag, New York, 2000. *В этом эссе вы найдете введение в замечательное собрание статей об исторически значимых системах.*

[B+72] «TENEX, A Paged Time Sharing System for the PDP-10» by D. Bobrow, J. Burchfiel, D. Murphy, R. Tomlinson. CACM, Volume 15, Number 3, March 1972. *TENEX вобрала в себя многие механизмы, встречающиеся в современных операционных системах; почитайте о ней – и узнаете, сколько инноваций было придумано еще в начале 1970-х годов.*

[B75] «The Mythical Man-Month» by F. Brooks. Addison-Wesley, 1975. *Классическая книга по программной инженерии, очень рекомендуем прочитать*[1].

[BOH10] «Computer Systems: A Programmer's Perspective» by R. Bryant and D. O'Hallaron. Addison-Wesley, 2010. *Еще одно великолепное введение в компьютерные системы. Немного пересекается с этой книгой, поэтому, если хотите, можете пропустить несколько последних глав или прочитайте их, чтобы познакомиться с другим взглядом на тот же самый материал. В конце концов, один из способов познакомиться с предметом – изучить как можно больше разных точек зрения, а затем сформулировать собственное мнение и мысли. Думать надо, думать!*

[G85] «The GNU Manifesto» by R. Stallman. 1985. www.gnu.org/gnu/manifesto.html. *Своим успехом Linux, без сомнения, в огромной степени обязана великолепному компилятору gcc и другому программному обеспечению, появившемуся в результате усилий по созданию фонда GNU, который возглавил Ричард Столлмен – провидец в области ПО с открытым исходным кодом. А в этом манифесте изложены его мысли на эту тему.*

[K+61] «One-Level Storage System» by T. Kilburn, D. B. G. Edwards, M. J. Lanigan, F. H. Sumner. IRE Transactions on Electronic Computers, April 1962. *В системе Atlas впервые было реализовано многое из того, что мы встречаем в современных системах. Однако эта статья – не самое лучшее чтение. Если у вас есть время для чтения только одной работы, то лучше обратиться к историческому обзору [L78].*

[L78] «The Manchester Mark I and Atlas: A Historical Perspective» by S. H. Lavington. Communications of the ACM, Volume 21:1, January 1978. *Изящно изложенная история ранних разработок компьютерных систем и первопроходческой*

[1] *Фредерик Брукс младший.* Мифический человеко-месяц. Питер, 2020.

роли Atlas. Конечно, можно было бы прочитать оригинальные статьи по Atlas, но эта работа содержит отличный обзор и открывает определенную историческую перспективу.

[O72] «The Multics System: An Examination of its Structure» by Elliott Organick. MIT Press, 1972. *Прекрасный обзор системы Multics. Изобилие блестящих идей, но при этом система замахнулась на слишком многое и потому в реальности так никогда и не работала. Классический пример того, что Фред Брукс назвал «эффектом второй системы» [B75].*

[PP03] «Introduction to Computing Systems: From Bits and Gates to C and Beyond» by Yale N. Patt, Sanjay J. Patel. McGraw-Hill, 2003. *Одно из наших любимых введений в компьютерные системы. Начав с транзисторов, проводит по всему пути к языку C; особенно хороши начальные главы.*

[RT74] «The Unix Time-Sharing System» by Dennis M. Ritchie, Ken Thompson. CACM, Volume 17: 7, July 1974. *Блестящий обзор Unix, написанный ее создателями в то время, когда эта система захватывала мир.*

[S68] «SDS 940 Time-Sharing System» by Scientific Data Systems. TECHNICAL MANUAL, SDS 90 11168, August 1968. *Да, техническое руководство – лучшее, что нам удалось найти. Но как же захватывает чтение этой документации по старой системе, когда понимаешь, как много было сделано еще в конце 1960-х годов. Одним из гениев, стоявших у истоков Berkeley Time-Sharing System (которая в итоге превратилась в систему SDS), был Батлер Лэмпсон, позже получивший премию Тьюринга за вклад в разработку вычислительных систем.*

[SS+10] «Membrane: Operating System Support for Restartable File Systems» by S. Sundararaman, S. Subramanian, A. Rajimwale, A. Arpaci-Dusseau, R. Arpaci-Dusseau, M. Swift. FAST '10, San Jose, CA, February 2010. *Что хорошо, когда пишешь заметки к собственным лекциям, – можно рекламировать свои исследования. Но эта статья действительно интересна – когда файловая система из-за ошибки «падает», Membrane автоматически перезапускает ее, так что ни приложения, ни прочие части системы не страдают.*

Домашнее задание

Большинство глав этой книги (а в конечном итоге так будет со всеми) заканчиваются домашним заданием. Выполнять их важно, поскольку так читатель приобретает практический опыт работы с изложенным в главе материалом.

Есть два типа домашних заданий. Первый основан на **эмуляции**. Эмулятор компьютерной системы – это простая программа, которая моделирует какие-то интересные части настоящей системы и формирует отчет о метриках, показывающих, как ведет себя система. Например, эмулятор жесткого диска отправляет серию запросов, моделирует время, необходимое диску для их обслуживания в предположении определенных характеристик производительности, а затем формирует отчет о средней задержке обработки запроса.

Эмуляция хороша тем, что позволяет исследовать поведение систем, обходя трудности, связанные с эксплуатацией реальной системы. На самом деле

так можно даже создать систему, не существующую в реальности (скажем, невообразимо быстрый жесткий диск), и изучать потенциальное влияние будущих технологий.

Конечно, у эмуляции есть и недостатки. По самой своей природе, эмуляция – это лишь приближение к реальной системе. Если упустить из виду какой-то важный аспект реального поведения, то результаты эмулятора окажутся бесполезны. Поэтому к результатам эмуляции всегда следует относиться с подозрением. В конце концов, по-настоящему важно, как система ведет себя в реальном мире.

Домашние задания второго типа предполагают взаимодействие с **реальным кодом**. Некоторые из них нацелены на измерения, другие требуют не слишком масштабной разработки и экспериментирования. В любом случае это лишь небольшие вылазки в обширный мир разработки системного кода на C в UNIX-системах, куда вам рано или поздно предстоит попасть. Чтобы дальше продвинуться в этом направлении, необходимы более крупные проекты, выходящие за рамки домашних заданий, поэтому мы настоятельно рекомендуем не ограничиваться домашними заданиями и выполнять проекты, чтобы закрепить приобретенные навыки. Некоторые проекты такого рода описаны на странице https://github.com/remzi-arpacidusseau/ostep-projects.

Для выполнения домашних заданий вам, вероятно, понадобится машина под управлением Linux, macOS или аналогичной системы. Должен быть также установлен компилятор C (например, **gcc**) и Python. Кроме того, вы должны знать, как писать код в каком-нибудь редакторе кода.