Ввод-вывод. Базовые понятия

Лекция

Ревизия: 0.2

Последняя модификация: Vlad Kovtun

Дата последней модификации: 03.04.2015

История изменений

09.03.2010 – Версия 0.1. Первичный документ. Ковтун В.Ю.

31.07.2014 – Версия 0.2. Внесены изменения в разделы Базовые понятия, Еще раз о прерываниях, замена рисунков. Ковтун В.Ю.

Содержание

Лекция 6. Ввод-вывод. Базовые понятия. Часть 1

Вопросы

1. Базовые понятия.
2. Принципы построения ПО ввода-вывода.
3. Программные уровни в ПО ввода-вывода.

Базовые понятия

Устройства ввода-вывода делятся на две категории:

- **Блочные устройства** - хранящие информацию в виде блоков фиксированного размера, причем у каждого блока имеется адрес. Обычно размеры блоков варьируются от 521 до 32 768 байт. **Важное свойство** блочного устройства состоит в том, что каждый его блок может быть прочитан независимо от остальных блоков. Наиболее распространенными блочными устройствами являются диски.
- **Символьные устройства** – принимает или предоставляет поток символов без какой-либо блочной структуры. Оно не является адресуемым и не выполняет операцию поиска. Принтеры, сетевые интерфейсные карты, «мыши» и большинство других устройств, не похожих на диски, можно рассматривать как символьные устройства.

Если приглядеться внимательнее, то окажется, что граница между блок-адресуемыми устройствами и устройствами, с отдельным блоком, которые нельзя адресоваться напрямую, не определена строго. Все согласны с тем, что диск является блок-адресуемым устройством, так как вне зависимости от текущего положения головки дисковода всегда можно переместить ее на определенный цилиндр и затем считать или записать отдельный блок с нужной дорожки. Рассмотрим теперь накопитель на магнитной ленте (магнитофон), применяемый для хранения резервных копий диска. На ленте хранится последовательность блоков. Если магнитофону дать команду прочитать блок N, он всегда может перемотать ленту и начать читать блоки, пока не дойдет до запрашиваемого блока N. Эта операция подобна поиску блока на диске с той лишь разницей, что она занимает значительно больше времени. Кроме того, в зависимости от накопителя и формата хранящихся на нем данных, может оказаться возможной или невозможной запись отдельного произвольного блока в середине ленты.

Такая схема классификации не совершенна. Некоторые устройства просто не попадают ни в одну из категорий. Например, часы не являются блок-адресуемыми. Они также не формируют и не принимают символьных потоков. Вся их работа состоит в инициировании прерываний в строго определенные моменты времени. Экраны отображения памяти также не втискиваются в рамки этой модели. И все же **модель блочных и символьных устройств** является настолько общей, что может использоваться в качестве основы для достижения независимости от устройств некоторого ПО ОС, имеющего дело с вводом-выводом. Например, файловая система имеет дело с абстрактными блочными устройствами, а зависимую от устройств часть составляет ПО низкого уровня.

Следует также отметить существенные различия между устройствами ввода-вывода, принадлежащими к разным классам, и в рамках каждого класса. Эти различия касаются следующих характеристик:

- скорость передачи данных (различия на несколько порядков);
- применение. Каждое действие, поддерживаемое устройством, оказывает влияние на программное обеспечение и стратегии операционной системы (например, диск, используемый для хранения файлов или для страниц виртуальной памяти, требует различного программного обеспечения);
- сложность управления. Для принтера требуется относительно простой интерфейс управления, для диска – намного сложнее. Влияния этих отличий на ОС сглаживается усложнением контроллеров ввода-вывода;
- единицы передачи данных. Данные могут передаваться блоками или потоками байтов или символов;
- представления данных. Различные устройства используют разные схемы кодирования данных, включая разную кодировку символов и контроль четности;

- условия ошибки. Природа ошибок, способ сообщения о них, их последствия и возможные ответы резко отличаются при переходе от одного устройства к другому.

Контроллеры устройств

Устройства ввода-вывода обычно состоят из:

- механической части;
- электронной части.

Часто эти части можно разделить для придания модели более модульного и общего вида. Электронный компонент устройства называется **контроллером устройства или адаптером**. В ПК он часто принимает форму печатной платы, вставляемой в слот расширения. Механический компонент находится в самом устройстве. Такая организация показана на Рис. 1.

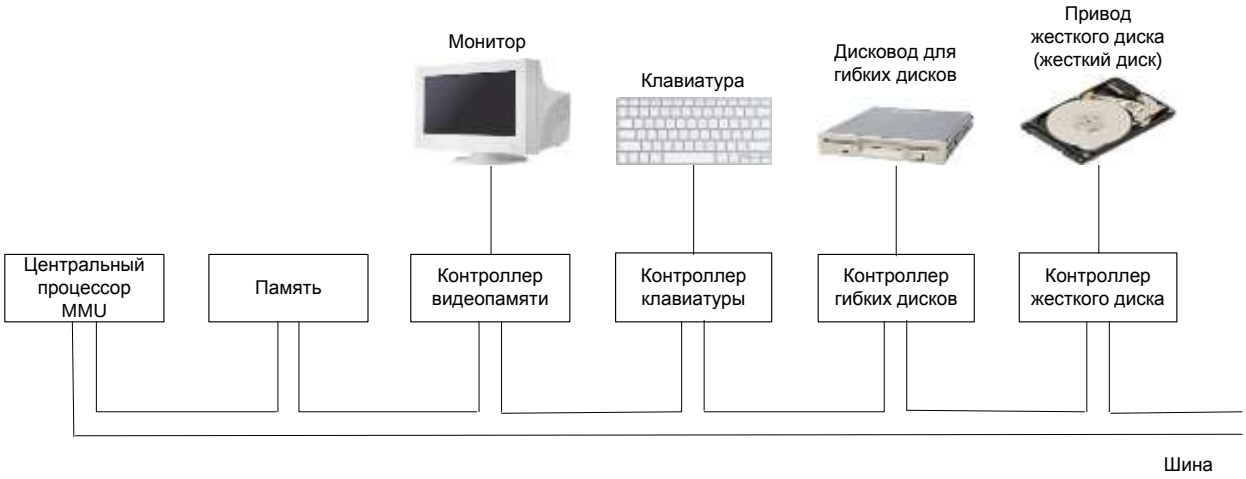

Рис. 1. Некоторые компоненты ПК

Плата контроллера обычно снабжается разъемом, к которому может быть подключен кабель, ведущий к самому устройству. Многие контроллеры способны управлять двумя, более идентичными устройствами. Если **интерфейс** между контроллером и устройством стандартизированы, то есть официальным стандартом ANSI, IEEE или ISO либо фактическим стандартом, тогда различные компании могут выпускать отдельно контроллеры и устройства, удовлетворяющие данному интерфейсу. Так, многие компании производят жесткие диски, соответствующие интерфейсу SAS, SATA, eSATA, SATA II, IDE или SCSI.

Интерфейс между устройством и контроллером часто является интерфейсом очень низкого уровня. Например, какой-нибудь жесткий диск может быть отформатирован по 256 секторов на дорожку, с размером секторов по 512 байт. В действительности с диска в контроллер поступает последовательный поток битов, начинающийся с заголовка сектора (преамбулы), за которым следует 4096 бит в секторе, и, наконец, контрольная сумма, также называемая **кодом исправления ошибок (ECC, Error-Correcting Code)**. Заголовок сектора записывается на диск во время форматирования. Он содержит номера цилиндра и сектора, размер сектора, информацию синхронизации и т. п.

Работа контроллера заключается в конвертировании последовательного потока битов в блок байтов и выполнение коррекции ошибок, если это необходимо. Обычно байтовый блок собирается бит за битом в буфере контроллера. Затем проверяется контрольная сумма блока, и если она совпадает с указанной в заголовке сектора, блок объявляется считанным без ошибок, после чего он копируется в RAM.

Видеоадаптер также работает как **бит-последовательное устройство**, на таком же низком уровне. Он считывает в памяти байты, содержащие символы, которые следует отобразить, и формирует сигналы, используемые для модуляции луча электронной трубки, заставляющие ее выводить изображение на экран. Видеоадаптер также формирует сигналы, управляющие горизонтальным и вертикальным возвратом электронного луча. Если бы ни контроллер, программисту пришлось бы управлять

перемещениями аналогового электронного луча. В действительности же ОС всего лишь инициализирует контроллер, задавая небольшое число параметров, таких как количество символов или пикселов в строке и число строк на экране, а всю тяжелую работу по управлению передвижениями электронного луча по экрану выполняет контроллер.

Отображаемый на адресное пространство памяти ввод-вывод

У каждого контроллера есть несколько регистров, с помощью которых с ним может общаться CPU. При помощи записи в эти регистры ОС велит устройству предоставить данные, принять данные, включиться или выключиться и т. п. Читая из этих регистров, ОС может узнать состояние устройства, например, готово ли оно к приему новой команды и т. д.

Помимо управляющих регистров, у многих устройств есть буфер данных, из которого ОС может читать данные, а также писать данные в него. Например, для отображения пикселов на экране данные обычно помещаются в видеопамять, являющуюся, по сути, буфером данных, доступным ОС и другим программам для чтения и записи.

Существует два альтернативных способа реализации доступа к управляющим регистрам и буферам данных устройств ввода-вывода:

Первый вариант заключается в том, что каждому управляющему регистру назначается номер порта ввода-вывода, 8- или 16-разрядное целое число. При помощи такой специальной команды CPU, как

```
IN REG, PORT
```

CPU может прочитать управляющий регистр устройства из порта `PORT` в регистр CPU `REG`. Аналогично с помощью команды

```
OUT PORT, REG
```

CPU может записать содержимое своего регистра `REG` в управляющий регистр устройства через порт `PORT`.

При такой схеме адресные пространства ОЗУ (RAM) и устройств ввода-вывода не пересекаются, как видно из Рис. 2(а).

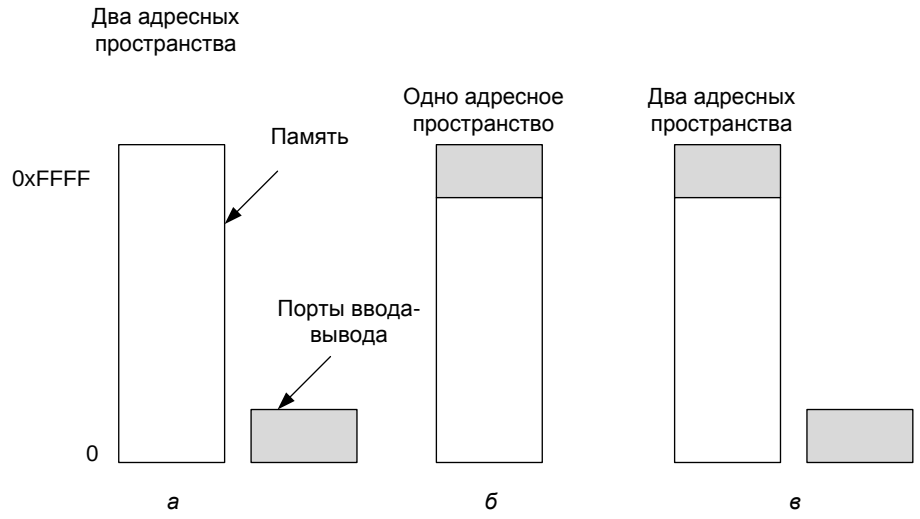

Рис. 2. Раздельные адресные пространства (а); отображаемый на адресное пространство памяти ввод-вывод (б); гибрид (в)

Второй подход, состоит в отображении всех управляющих регистров периферийных устройств на адресное пространство памяти, как показано на Рис. 2(б). Каждому управляющему регистру назначался уникальный адрес в памяти. Такая система называется **отображаемым на адресное пространство памяти вводом-выводом**. Обычно для регистров устройств отводятся адреса на вершине адресного пространства. Также существуют различные гибридные схемы, с отображаемыми на адресное пространство памяти буферами данных и отдельными портами ввода-вывода, Рис. 2(в). Эта схема довольно широко применяется, например, в совместимых с IBM PC компьютерах на базе CPU 0x86, в которых, помимо портов ввода-вывода с номерами от

0 до 64 K, адресное пространство RAM от 640 K до 1 M зарезервировано под буферы данных устройств ввода-вывода.

Как работают все эти схемы? Во всех случаях, когда CPU хочет прочитать слово данных либо из памяти, либо из порта ввода-вывода, он выставляет нужный адрес на адресную шину, после чего выставляет сигнал READ на управляющую шину. Вторая сигнальная линия позволяет отличить обращение к памяти от обращения к порту. В зависимости от состояния этой линии шины управления на запрос CPU реагирует устройство (контроллер) ввода-вывода или память. Если пространство адресов общее, Рис. 2(б), то каждый модуль памяти и каждое устройство ввода-вывода сравнивает выставленный на шину адрес с обслуживаемым им диапазоном адресов. Если выставленный на шину адрес попадает в этот диапазон, то соответствующе устройство реагирует на запрос CPU. Поскольку выделенные внешним устройствам адреса удаляются из памяти, память не реагирует на них и конфликта адресов не происходит.

Обе схемы обращения к контроллерам имеют свои сильные и слабые стороны. Начнем с **достоинств отображаемого на адресное пространство памяти ввода-вывода**. **Во-первых**, при такой схеме для обращения к устройствам ввода-вывода не требуются специальные команды CPU, такие как `IN` и `OUT`. В результате программу, общающуюся с таким устройством, можно написать целиком на языке C или C++, без вставок на ассемблере или обращений к подпрограммам, написанным на ассемблере, то есть без дополнительных накладных расходов.

Во-вторых, при отображении регистров ввода-вывода на память не требуется специального механизма защиты от пользовательских процессов, пытающихся обращаться к внешним устройствам. Все, что нужно сделать ОС, — это исключить ту часть адресного пространства, на которую отображаются управляющие регистры устройств ввода-вывода из адресного пространства пользователей. Более того, если управляющие регистры различных устройств ввода-вывода отображаются на различные страницы памяти, ОС может предоставить доступ к различным страницам различным пользователям, таким образом, предоставляя пользователям доступ к одним устройствам и запрещая доступ к другим. Для этого нужно всего лишь включить номер соответствующей страницы памяти в карту памяти нужного пользователя. В результате такая схема позволяет разместить драйверы различных устройств в различных адресных пространствах, тем самым не только уменьшая размер ядра, но и удерживая драйверы от вмешательства в дела друг друга.

В-третьих, при отображении регистров ввода-вывода на память каждая команда CPU, обращающаяся к памяти, может с тем же успехом обращаться к управляющим регистрам устройства. Например, если у CPU есть в наборе команд инструкция CMP, проверяющая содержимое некоего слова в памяти (регистре) на равенство с неким значением. Управляющий регистр, равный 0, будет означать, например, готовность данного устройства к приему новой команды. Программа на ассемблере может выглядеть следующим образом:

```
MOV ES, 0B800h
LOOP: MOV AX, WORD PTR [ES]   // сравнить содержимое порта 4 с нулем
      CMP AX, 0               // сравнить содержимое адреса с нулем
      JZ READY                // если он равен 0. идти на метку READY
      JMP LOOP                // в противном случае продолжать опрос порта
READY:
```

Если отображения регистров ввода-вывода на память нет, управляющий регистр устройства должен быть сначала считан в регистр CPU, а уже затем сравнен с 0, что требует двух команд CPU вместо одной. Для приведенного выше цикла добавление четвертой команды может слегка снизить (все зависит от конкретных CPU, конечно) скорость реакции драйвера на появление признака готовности устройства.

В разработке компьютеров практически у любого решения есть как положительные, так и отрицательные стороны. Отображение регистров ввода-вывода на память также **обладает недостатками**. **Во-первых**, в большинстве современных компьютеров применяется кэширование памяти. Кэширование управляющих регистров привело бы просто к катастрофе. Поясним это утверждение на уже приведенном выше примере программы, опрашивающей в цикле адрес `[ES]`. При первом обращении к этому адресу считалось бы верное значение порта, но это значение сохранилось бы в кэше. Все последующие обращения к этому адресу, на следующих итерациях цикла, просто читали бы значение, сохраненное в кэше, и никогда не обращались бы к реальному

устройству. Таким образом, программа никогда не вышла бы из цикла ожидания готовности, так как при кэшировании регистров устройств она просто не смогла бы узнать об изменении управляющего регистра.

Чтобы не допустить такой ситуации, необходима специальная аппаратура, способная выборочно запрещать кэширование, например, в зависимости от номера страницы памяти, к которой обращается CPU. Таким образом, отображение регистров ввода-вывода на память увеличивает сложность аппаратуры и ОС, которой приходится управлять избирательным кэшированием.

Во-вторых, при едином адресном пространстве все модули памяти и все устройства ввода-вывода должны изучать все обращения CPU к памяти, чтобы определить, на которые им следует реагировать. Если у компьютера одна общая шина, Рис. 3(а), реализовать подобный просмотр всех обращений к памяти всеми устройствами несложно.

Однако в конструкции современных ПК наблюдается тенденция в сторону использования выделенной высокоскоростной шины (Рис. 3(б)), архитектурной особенности, кстати, уже давно применявшейся в мэйнфреймах. Эта шина предназначена для увеличения скорости обмена данными между CPU и RAM, чему в архитектуре общей шины сильно мешали медленные устройства ввода-вывода. В компьютерах на базе CPU i486 таких внешних шин целых три (шина памяти, PCI и ISA).

Сложность применения выделенной шины памяти на машинах с отображением регистров ввода-вывода на память состоит в том, что у устройств ввода-вывода нет способа увидеть адреса памяти, выставляемые CPU на эту шину, следовательно, они не могут реагировать на такие адреса. Поэтому, чтобы отображение регистров ввода-вывода могло работать на системах с несколькими шинами, необходимы специальные меры. **Один способ** решения этой проблемы состоит в том, что сначала все обращения к памяти посылаются CPU по выделенной быстрой шине напрямую памяти (чтобы не снижать производительности). Если память не может ответить на эти запросы, CPU пытается сделать это еще раз по другим шинам. Такое решение работоспособно, но требует дополнительного увеличения сложности аппаратуры.

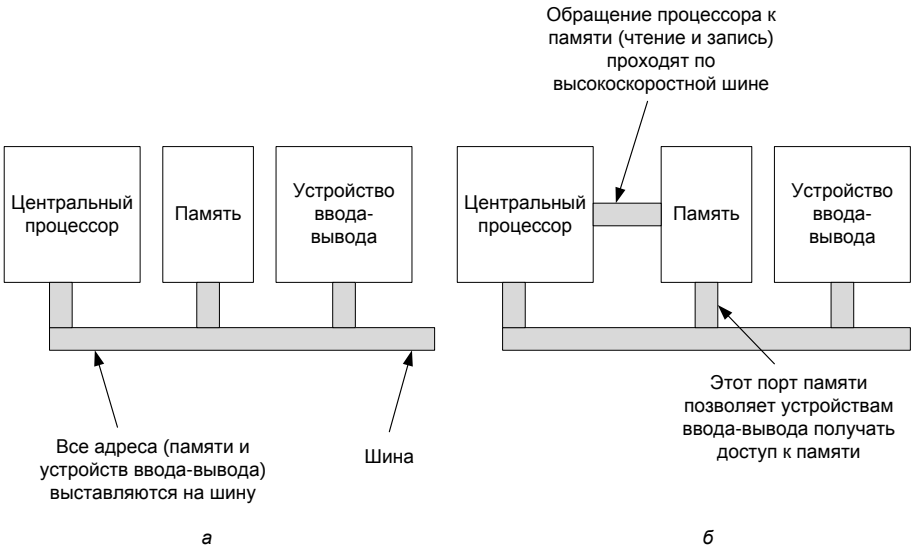

Рис. 3. Архитектура с одной шиной (а); архитектура памяти с двумя шинами (б)

Второе возможное решение заключается в установке на шину памяти специального следящего устройства, передающего все адреса потенциально заинтересованным устройствам ввода-вывода. Проблема, однако, р том, что устройства ввода-вывода могут просто не успеть обработать эти запросы с той же скоростью, что и память.

Третье решение, используемое в компьютерах на базе CPU i486, состоит в фильтрации адресов микросхемой моста PCI. Эта микросхема содержит регистры диапазона, заполняемые во время загрузки компьютера. Например, диапазон адресов от 640 К до 1 М может быть помечен как не относящийся к памяти. Все адреса, попадающие в подобный диапазон, передаются не памяти, а на шину PCI. Недостаток этой схемы состоит в необходимости принятия во время загрузки решения о том, какие адреса не являются адресами памяти. Итак, у каждой схемы есть свои достоинства и недостатки, так что компромиссы и уступки неизбежны.

Прямой доступ к памяти (DMA)

Независимо от того, отображаются ли регистры или буферы ввода-вывода на память или нет, CPU необходимо как-то адресоваться к контроллерам устройств для обмена данными с ними. CPU может запрашивать данные от контроллера ввода-вывода по одному байту, но подобная организация обмена данными крайне неэффективна, так как расходует огромное количество времени CPU. Поэтому на практике часто применяется другая схема, называемая **доступом к памяти (DMA - direct memory access)**.

ОС может воспользоваться DMA только при наличии аппаратного DMA-контроллера, который есть у большинства систем. Иногда DMA-контроллер интегрируется в другие контроллеры, например в дисковый контроллер, но такой дизайн требует оснащения DMA-контроллерами каждого периферийного устройства. Как правило, DMA-контроллер, устанавливаемый на материнской плате, обслуживает запросы по передаче данных нескольких различных устройств ввода-вывода, часто на конкурентной основе.

Где бы он ни располагался физически, DMA-контроллер может получать доступ к системной шине независимо от CPU, как показано на Рис. 4. Он содержит несколько регистров, доступных CPU для чтения и записи. К ним относятся регистр адреса памяти, счетчик байтов и один или более управляющих регистров. Управляющие регистры задают, какой порт ввода-вывода должен быть использован, направление переноса данных (чтение из устройства ввода-вывода или запись в него), единицу переноса (осуществлять перенос данных побайтно или пословно), а также число байтов, которые следует перенести за одну операцию.

Чтобы понять, как работает DMA, познакомимся сначала с тем, как происходит чтение с диска при отсутствии DMA. Сначала контроллер считывает с диска блок (один или несколько секторов) последовательно, бит за битом, пока весь блок не окажется во внутреннем буфере контроллера. Затем контроллер проверяет контрольную сумму, чтобы убедиться, что при чтении не произошло ошибки. После этого контроллер инициирует прерывание. Когда ОС начинает работу, она может прочитать блок диска побайтно или пословно, в цикле сохраняя считанное слово или байт в RAM.

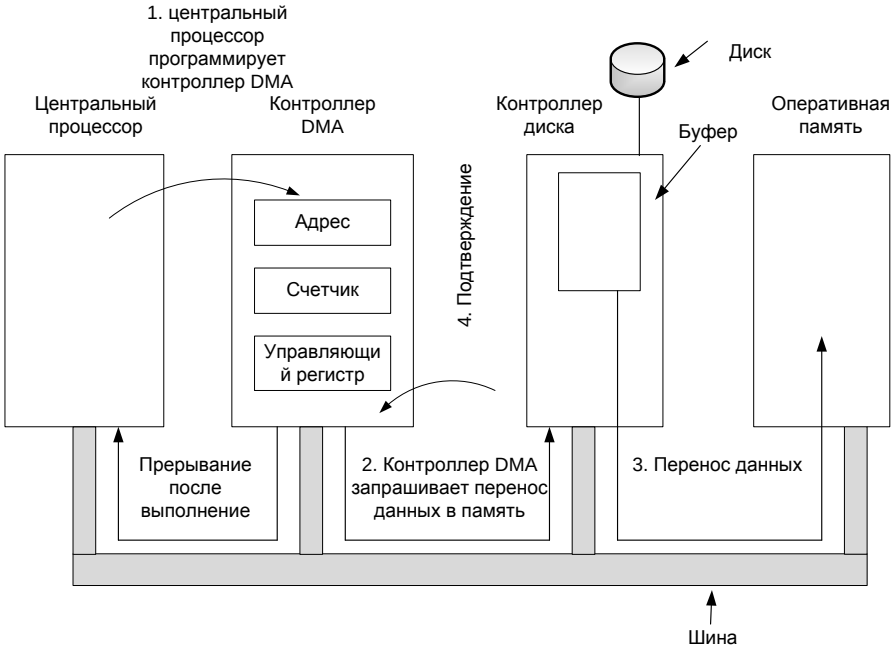

Рис. 4. Работа DMA-контролера

При использовании DMA процедура совершенно другая. Сначала CPU программирует DMA-котроллер, устанавливая его регистры и указывая, какие данные и куда следует переместить (шаг 1 на Рис. 4).

Затем CPU дает команду дисковому контроллеру прочитать данные во внутренний буфер и проверить контрольную сумму. Когда данные получены и проверены контроллером диска, DMA может начинать работу.

DMA-контроллер начинает перенос данных, посылая дисковому контроллеру по шине запрос чтения (шаг 2). Этот запрос чтения выглядит как обычный запрос чтения, так что контроллер диска даже не знает, пришел ли он от CPU или от контроллера DMA. Обычно адрес памяти уже находится на адресной шине, так что контроллер диска всегда знает, куда следует переслать следующее слово из своего внутреннего буфера. Запись в память является еще одним стандартным циклом шины (шаг 2). Когда запись закончена, контроллер диска также по шине посылает сигнал подтверждения контроллеру DMA (шаг 4), Затем контроллер DMA увеличивает используемый адрес памяти и уменьшает значение счетчика байтов. После этого шаги со 2-го по 4-й повторяются, пока значение счетчика не станет равно нулю. По завершении цикла копирования контроллер DMA инициирует прерывание CPU, сообщая ему таким образом, что перенос данных завершен. ОС не нужно копировать блок диска в память. Он уже находится там.

Контроллеры DMA значительно различаются по степени своей сложности. Самые простые из них за один раз выполняют одну операцию переноса данных, как описывалось выше. Более сложные контроллеры могут выполнять сразу несколько подобных операций. У таких контроллеров несколько каналов, каждый из которых управляется своим набором внутренних регистров. CPU начинает с того, что загружает в эти регистры соответствующие параметры. Все операции переноса данных должны выполняться с различными устройствами ввода-вывода. После переноса каждого слова данных (шаги 2-4 на Рис. 4) контроллер DMA решает, какое устройство будет им обслужено следующим. Этот выбор может производиться циклически или при помощи приоритетной схемы, предоставляющей одним устройствам преимущество по сравнению с другими. Одновременно несколько запросов могут дожидаться исполнения, при условии, что существует способ однозначно отличить подтверждения различных устройств. Часто с этой целью для каждого канала DMA используются различные линии подтверждения.

Многие шины могут работать в двух режимах:

- в пословном;
- в поблочном.

Некоторые контроллеры DMA также могут функционировать в обоих режимах. В пословном режиме процедура выглядит так, как описывалось выше: контроллер DMA выставляет запрос на перенос одного слова и получает его. Если CPU также нужна эта шина, ему приходится подождать. Этот механизм называется **захватом цикла (cycle stealing)**, потому что контроллер устройства периодически «подкрадывается» и забирает случайный цикл шины у CPU, слегка его тормозя. В блочном режиме контроллер DMA велит устройству занять шину, сделать серию пересылок и отпустить шину. Такой способ действий называется **пакетным режимом**. Он более эффективен, чем захват цикла, поскольку занятие шины требует времени, а в пакетном режиме эта процедура выполняется всего один раз для передачи целого блока данных. **Недостатком этого метода** является то, что при переносе большого блока данных он может заблокировать CPU и другие устройства на существенный промежуток времени.

В обсуждавшейся модели, иногда называемой **сквозным режимом**, контроллер DMA велит контроллеру устройства переслать данные напрямую в RAM. В некоторых DMA-контроллерах используется также режим, при котором контроллер устройства посылает слово данных контроллеру DMA, который затем выставляет на шину еще один запрос для передачи этого слова туда, куда его нужно передать. При такой схеме требуется лишний цикл шины на передачу каждого слова, зато такая схема обладает большей гибкостью, так как также позволяет выполнять копирование с устройства на устройство, минуя память, и даже из памяти в память. (Для этого нужно сначала дать команду чтения из памяти, а затем команду записи в память, но по другому адресу.)

Большинство контроллеров DMA используют для передачи данных физические адреса памяти. Чтобы использовать физические адреса памяти, ОС должна преобразовать виртуальный адрес буфера памяти в физический и записать этот физический адрес в адресный регистр контроллера DMA. В некоторых контроллерах DMA применяется альтернативная схема, при которой в контроллер DMA записывается сразу виртуальный адрес, В этом случае контроллер DMA должен использовать менеджер памяти MMU для преобразования адреса. Виртуальный адрес может быть выставлен на адресную шину только в том случае, когда MMU является частью памяти (что возможно, но редко), а не частью CPU.

До начала операции DMA диск сначала считывает данные в свой внутренний буфер. Почему контроллер не помещает данные прямо в RAM, по мере получения их с диска? Другими словами, зачем ему нужен внутренний буфер? Тому есть две причины:

Во-первых, при помощи внутренней буферизации контроллер диска может проверить контрольную сумму до начала переноса данных в память. Если контрольные суммы не совпадают, формируется сигнал об ошибке и перенос данных не производится.

Во-вторых, дело в том, что как только началась операция чтения с диска, биты начинают поступать с постоянной скоростью, независимо от того, готов контроллер диска их принимать или нет. Если контроллер диска попытается писать эти данные напрямую в память, ему придется делать это по системной шине. Если при передаче очередного слова шина окажется занятой каким-либо другим устройством (например, использующим ее в пакетном режиме), контроллеру диска придется ждать. Если следующее слово с диска прибудет раньше, чем контроллер успеет сохранить предыдущее, контроллер либо потеряет предыдущее слово, либо ему придется сохранять его где-либо еще. Таким образом, необходимость внутреннего буферирования становится очевидной. При наличии внутреннего буфера контроллеру диска шина не нужна до тех пор, пока не начнется операция DMA. В результате устройство контроллера диска оказывается проще, так как при операции DMA пересылки данных параметр времени не является критичным.

DMA используется не во всех компьютерах. **Главный аргумент против использования** DMA состоит в том, что CPU обычно значительно превосходит DMA-контроллер по скорости и может выполнить ту же работу значительно быстрее (если только скорость ограничена не быстродействием устройства ввода-вывода). При отсутствии другой работы у CPU заставлять быстрый CPU ждать, пока медленный контроллер DMA выполнит свою работу, бессмысленно. Кроме того, компьютер без контроллера DMA, с CPU, выполняющим всю работу программно, оказывается дешевле, что крайне важно в производстве компьютеров нижней ценовой категории (например, встроенных).

Еще раз о прерываниях

Структура прерываний типичного ПК проиллюстрирована на Рис. 5. На аппаратном уровне прерывания работают следующим образом. Когда устройство ввода-вывода заканчивает свою работу, оно инициирует прерывание (при условии, что прерывания разрешены ОС). Для этого устройство выставляет сигнал на выделенную устройству специальную линию шины. Этот сигнал распознается микросхемой контроллера прерываний, расположенной на материнской плате. Контроллер прерываний принимает решение о дальнейших действиях.

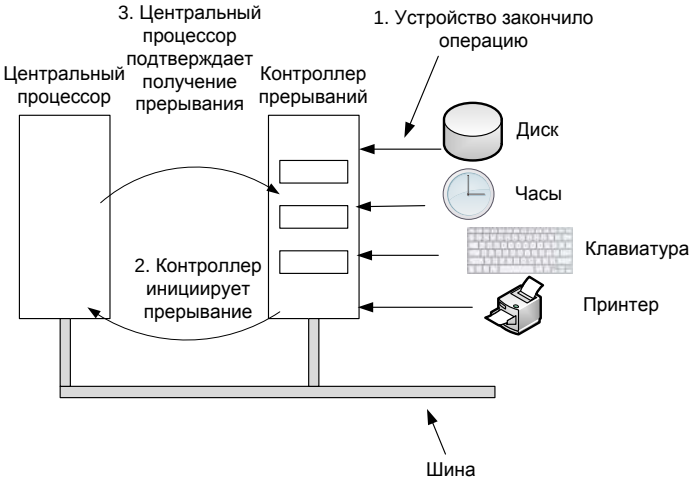

Рис. 5. Схема прерываний в компьютере. Соединения между устройствами и Контроллером прерываний в действительности являются специальными линиями шины, а не выделенными проводами

При отсутствии других необработанных запросов прерывания контроллер прерываний обрабатывает прерывание немедленно. Если прерывание уже обрабатывается, и в это время приходит запрос от другого устройства по линии с более низким приоритетом, то

новый запрос просто игнорируется. В этом случае устройство продолжает удерживать сигнал прерывания на шине до тех пор, пока оно не будет обслужено CPU.

Для обработки прерывания контроллер выставляет на адресную шину номер устройства, требующего к себе внимания, и устанавливает сигнал прерывания на соответствующий контакт CPU.

Этот сигнал заставляет CPU приостановить текущую работу и начать выполнять обработку прерывания. Номер, выставленный на адресную шину, используется в качестве индекса в таблице, называемой **вектором прерывании**, из которой извлекается новое значение счетчика команд. Новый счетчик команд указывает на начало соответствующей процедуры обработки прерывания. Обычно с этого места аппаратные и эмулированные прерывания используют один и тот же механизм и часто пользуются одним и тем же вектором. Расположение вектора может быть либо жестко прошито на аппаратном уровне, либо, наоборот, располагаться в произвольном месте памяти, на которое указывает специальный регистр CPU, загружаемый ОС.

Вскоре после начала своей работы процедура обработки прерываний подтверждает получение прерывания, записывая определенное значение в порт контроллера прерываний. Это подтверждение разрешает контроллеру издавать новые прерывания. Благодаря тому, что CPU откладывает выдачу подтверждения до момента, когда он уже готов к обработке нового прерывания, удается избежать ситуации состязаний при появлении почти одновременных прерываний от нескольких устройств. Следует упомянуть, что на некоторых старых компьютерах нет микросхемы, централизованного контроллера прерываний, поэтому контроллер каждого устройства выставляет свое собственное прерывание.

Аппаратура всегда, прежде чем начать процедуру обработки прерывания, сохраняет определенную информацию. Сохраняемая информация и место ее хранения широко варьируются в зависимости от CPU. Как минимум сохраняется счетчик команд, что позволяет продолжить выполнение прерванного процесса. Другая крайность представляет собой сохранение всех программно доступных регистров и большого количества внутренних регистров CPU.

Возникает вопрос, где нужно хранить эту информацию. Можно поместить ее во внутренние регистры, значение которых ОС может считывать по мере надобности. Но при этом возникает проблема, не позволяющая дать подтверждение контроллеру прерываний до тех пор, пока не будет считана вся потенциально важная информация, поскольку следующее прерывание при сохранении состояния перепишет все внутренние регистры. Такая стратегия приводит к продолжительным простоям, в течение которых запрещены прерывания, и к возможным потерям сигналов прерываний и потерям данных.

Поэтому большинство CPU сохраняют информацию в стеке. Но этот подход также имеет свои проблемы. Сначала возникает вопрос: в чьем стеке хранить данные? Если использовать текущий стек, то он может быть стейком пользовательского процесса. Указатель стека может даже содержать недопустимое значение, что приведёт к фатальной ошибке при попытке оборудования записать несколько слов по адресу, на который он указывает. Он также может указывать на конец страницы. После нескольких записей может произойти выход за ее пределы, и будет сгенерирована ошибка отсутствия страницы. Возникновение этой ошибки во время обработки аппаратного прерывания создаёт весьма серьёзную проблему: где сохранить состояние, чтобы обработать ошибку отсутствия страницы?

При использовании стека ядра появляется намного больше шансов на то, что указатель стека содержит допустимое значение и указывает на фиксированную страницу. Но переключение в режим ядра может потребовать изменения контекста MMU и, вероятно, сделает недействительной большую часть или даже все содержимое кэша и TLB. Их статистическая или динамическая перезагрузка увеличит время обработки прерывания и приведёт к пустой трате времени CPU.

Точные и неточные прерывания

Прерывание, оставляющее машину в строго определенном состоянии, называется **точным прерыванием**. У такого прерывания четыре следующих свойства:

1. Счетчик команд (IC, Instruction Counter) сохраняется в известном месте.

2. Все команды до той, на которую указывает счетчик команд, выполнены полностью.

3. Ни одна команда после той, на которую указывает счетчик команд, не была выполнена.

4. Состояние команды, на которую указывает счетчик команд, известно.

Следует обратить внимание, что здесь не накладывается запрета на начало выполнения команд после той, на которую указывает счетчик команд. Утверждается лишь, что все изменения с памятью и регистрами CPU, произведенные благодаря началу выполнения этих инструкций, должны быть отменены прежде, чем начнется обработка сигнала прерывания CPU. Разрешается выполнение команды, на которую указывает счетчик команд. Также допускается ее невыполнение. Однако должно быть известно, выполнена она или нет. Часто случается, если прерывание пришло от устройства ввода-вывода, что выполнение команды, на которую указывает счетчик команд, еще и не начиналось. Однако если прерывание было эмулировано или вызвано обращением к отсутствующей странице, тогда счетчик команд обычно указывает на команду, вызвавшую прерывание.

Прерывание, не удовлетворяющее данным требованиям, называется **неточным прерыванием**. Такое прерывание крайне портит жизнь программистам, пишущим ОС, которым приходится в таком случае выяснять, что случилось и чему еще предстоит случиться. Машины с неточным прерыванием обычно в случае прерывания выгружают в стек огромное количество данных, чтобы дать ОС возможность определить, что происходило в этот момент. Сохранение больших объемов данных в памяти при каждом прерывании сильно замедляет вход в процедуру обработки прерывания, а восстановление после прерывания усложняется еще больше. Все это приводит к нелепой ситуации, в которой сверхбыстрый суперскалярный CPU оказывается непригоден для задач реального времени из-за своих страшно медленных прерываний.

Некоторые компьютеры спроектированы таким образом, что одни типы прерываний (аппаратных и эмулированных) оказываются точными, тогда как другие — неточными. Например, совсем не плохо, если прерывания от устройств ввода-вывода будут точными, а эмулированные прерывания и прерывания, вызванные программными ошибками, будут неточными, так как последние не требуют возобновления прерванных процессов. В некоторых машинах имеется специальный бит, установив который можно все прерывания сделать точными. Недостатком установки такого бита является то, что он вынуждает CPU тщательно регистрировать свои действия и сохранять значения регистров в специальных теневых регистрах, обеспечивая, таким образом, возможность произвести точное прерывание в любой момент времени. Естественно, все эти накладные расходы заметно снижают производительность.

Принципы программного обеспечения ввода-вывода

Перейдем теперь от рассмотрения аппаратуры ввода-вывода к знакомству с ПО ввода-вывода. Сначала мы познакомимся с целями ПО ввода-вывода, а затем с различными способами выполнения операций ввода-вывода сточки зрения ОС.

Задачи программного обеспечения ввода-вывода

Ключевая концепция разработки ПО ввода-вывода известна как **независимость от устройств**. Эта концепция означает возможность написания программ, способных получать доступ к любому устройству ввода-вывода, без предварительного указания конкретного устройства. Например, программа, читающая данные из входного файла, должна с одинаковым успехом работать с файлом на дискете, жестком диске или компакт-диске. При этом не должны требоваться какие-либо изменения в программе. Например, должна быть возможность дать команду вроде

```
sort <input >output
```

и эта команда должна работать, независимо от того, что указано в качестве входного устройства — гибкий диск, IDE-диск, SCSI-диск или клавиатура. В качестве выходного устройства также с равным успехом может быть указан экран, файл на любом диске или принтер. Все проблемы, связанные с отличиями этих устройств, должна решать ОС.

Тесно связан с концепцией независимости от устройств принцип единообразного именования. Имя файла или устройства должно быть просто текстовой строкой или целым числом и никоим образом не зависеть от физического устройства. В системе UNIX все диски могут быть произвольным образом интегрированы в иерархию файловой системы, так что пользователю не обязательно знать, какое имя какому устройству соответствует. Например, гибкий диск может быть смонтирован поверх каталога /usr/ast/backup, так что копирование файла в каталог /usr/ast/backup/monday

автоматически приведет к копированию файлов на гибкий диск. Таким образом, все файлы и устройства адресуются одним и тем же способом: по имени пути.

Другим важным аспектом ПО ввода-вывода является **обработка ошибок**. Ошибки должны обрабатываться как можно ближе к аппаратуре. Если контроллер обнаружил ошибку чтения, он должен попытаться по возможности исправить эту ошибку сам. Если он не может это сделать, тогда эту ошибку должен обработать драйвер устройства, возможно, попытавшись прочитать этот блок еще раз. Многие ошибки бывают временными, как, например, ошибки чтения, вызванные пылинками на читающих головках. Такие ошибки часто исчезают при повторной попытке чтения блока. Только если нижний уровень не может сам справиться с проблемой, о ней следует информировать верхний уровень. Во многих случаях восстановление после ошибок может осуществляться на нижнем уровне, прозрачно для верхних уровней, то есть так, что верхние уровни даже не будут знать о наличии ошибок.

Еще одним ключевым вопросом является **способ переноса данных**:

- Синхронный (блокирующий).
- Асинхронного (управляемого прерываниями).

Большинство операций ввода-вывода на физическом уровне являются асинхронными — CPU запускает перенос данных и отправляется заниматься чем-либо другим, пока не придет прерывание. Программы пользователя значительно легче написать, используя блокирующие операции ввода-вывода — после обращения к системному вызову `read` программа автоматически приостанавливается до тех пор, пока данные не появятся в буфере. Тем, чтобы операции ввода-вывода, в действительности являющиеся асинхронными, выглядели как блокирующие в программах пользователя, занимается ОС.

Еще одним аспектом ПО ввода-вывода является **буферизация**. Часто данные, поступающие с устройства, не могут быть сохранены сразу там, куда они в конечном итоге направляются. Например, когда пакет приходит по сети, ОС не знает, куда его поместить, пока не будет изучено его содержимое, для чего этот пакет нужно где-то временно сохранить. Кроме того, для многих устройств реального времени крайне важными оказываются параметры сроков поступления данных (например, для устройств воспроизведения цифрового звука), поэтому полученные данные должны быть помещены в выходной буфер заранее, чтобы скорость, с которой эти данные получаются из буфера воспроизводящей программой, не зависела от скорости заполнения буфера. Таким образом, удается избежать неравномерности воспроизведения звука. Буферизация включает копирование данных в значительных количествах, что часто является основным фактором снижения производительности операций ввода-вывода.

И последним понятием, которое следует упомянуть, является понятие **выделенных устройств и устройств коллективного использования**. С некоторыми устройствами ввода-вывода, такими как диски, может одновременно работать большое количество пользователей. При этом не должно возникать проблем, если несколько пользователей на одном и том же диске одновременно откроют файлы. Другие устройства, такие как накопители на магнитной ленте, должны предоставляться в монопольное владение одному пользователю, пока он не завершит свою работу с этим устройством. После этого накопитель может быть предоставлен другому пользователю. Если два или более пользователей одновременно станут писать вперемешку блоки на одну ленту, то ничего хорошего не получится. Введение понятия выделенных (монопольно используемых) устройств также привносит целый спектр проблем, например, как взаимоблокировки. Тем не менее ОС должна уметь управлять как устройствами общего доступа, так и выделенными устройствами, позволяя избегать различных потенциальных проблем.

Способы осуществления ввода-вывода

Существует три фундаментально различных способа осуществления операций ввода-вывода:

- Программный ввод-вывод.
- Управляемый прерываниями.
- Ввод-вывод с использованием DMA.

Далее, в этом разделе **рассмотрим первый способ (программный ввод-вывод)**. В следующих двух разделах познакомимся с остальными разновидностями ввода-вывода (с управляемым прерываниями вводом-выводом и вводом-выводом с использованием

DMA). Простейший вид ввода-вывода состоит в том, что всю работу выполняет CPU. Этот метод называется программным вводом-выводом.

Проще всего проиллюстрировать программный ввод-вывод на примере. Рассмотрим процесс пользователя, которому нужно напечатать на принтере строку из восьми символов «ABCDEFGH». Сначала он собирает эту строку в буфере в пространстве пользователя (Рис. 6(а)).

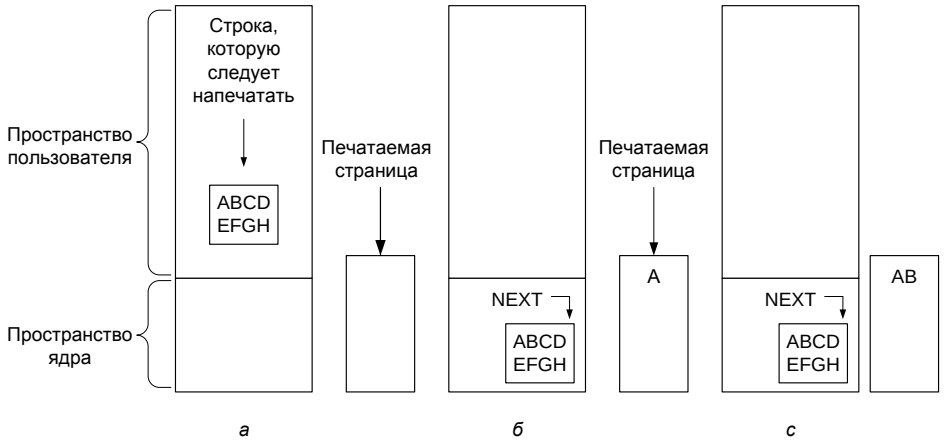

Рис. 6. Этапы печати строки

Затем, обращаясь к системному вызову, процесс пользователя получает принтер во временное пользование. Если принтер в данный момент оказывается занят другим процессом, обращение к системному вызову на открытие принтера завершится неудачей. Вызывающему процессу либо будет возвращен код ошибки, либо этот процесс будет блокирован до тех пор, пока принтер не освободится, в зависимости от ОС и параметров вызова. Получив принтер, процесс пользователя обращается к другому системному вызову, прося ОС распечатать строку на принтере.

ОС при этом обычно копирует содержимое буфера со строкой в некий массив, расположенный в пространстве ядра, где ей проще получить к этим данным доступ (поскольку ядру для получения доступа к пространству пользователя, возможно, придется изменять карту памяти). Затем она проверяет, доступен ли в данный момент принтер. Если нет, она ждет его освобождения. Как только принтер становится доступен, ОС копирует первый символ в регистр данных принтера, используя в данном примере отображение регистров устройств ввода-вывода на память. Это действие активизирует принтер. На бумаге этот символ может сразу не появиться, так как большинство принтеров буферизируют целую строку или даже страницу данных прежде, чем начать собственно печать. Однако на Рис. 6(б) видим, что первый символ напечатан, а указатель ОС установлен на следующий символ (B).

Напечатав первый символ на принтере, ОС проверяет, готов ли принтер к приему следующего символа. Обычно у принтера есть второй регистр, в котором можно прочитать его состояние. При записи символа в регистр данных принтер инвертирует бит готовности в статусном регистре. По окончании обработки полученного символа контроллером принтера бит готовности снова устанавливается, показывая, что принтер готов к приему следующего символа.

Итак, ОС ждет, когда принтер снова перейдет в состояние готовности. Когда это происходит, она печатает следующий символ, Рис. 6(в). Этот цикл продолжается до тех пор, пока не будет распечатана вся строка. После этого управление возвращается процессу пользователя.

Действия, выполняемые ОС, в виде программы на языке С продемонстрированы в ниже. Сначала данные копируются в ядро. Затем ОС входит в цикл, в котором на каждой итерации цикла печатает на принтере один символ. Существенный аспект программного ввода-вывода, ясно проиллюстрированный данным примером, состоит в том, что после печати каждого символа CPU в цикле опрашивает готовность устройства. Такое поведение CPU называется опросом или ожиданием готовности, а также активным ожиданием.

```
copy_from_user(buffer, p, count); // копирование байтов из buffer
                                  // (буфера ядра) в некоторый программный
```

```
                                    // буфер p в количестве count
for (i = 0; i < count; i++)         // цикл чтения символов
{
      while (*printer_status_reg != READY);  // цикл ожидания готовности
      *printer_data_reg = p[i];     // печень символа
}
return_to_user();
```

Программный ввод-вывод очень легко реализуется, но его существенный недостаток состоит в том, что **CPU занимается на все время операции ввода-вывода**. Даже если один символ «печатается» очень быстро, поскольку все, что нужно сделать принтеру — это поместить этот символ в свой внутренний буфер, принтер обычно не рассчитан на прием символов с той скоростью, с которой их может выдать CPU. Поэтому большую часть времени CPU проведет в ожидании готовности принтера, что является неэффективным использованием времени CPU. Такой подход вполне допустим в примитивных встроенных системах, в которых у CPU нет других задач; однако в более сложных, многозадачных системах такой подход неприемлем.

Управляемый прерываниями ввод-вывод

Рассмотрим теперь случай принтера, не буферизирующего символы, а печатающего их сразу по прибытии. Если принтер может печатать, скажем, 100 символов в секунду, то на печать каждого символа уходит 10 мс. Это значит, что после записи каждого символа в регистр данных принтера CPU должен ждать в цикле целых 10 мс, пока ему не позволят записать в регистр следующий символ. Этого времени более чем достаточно для переключения контекста и запуска другого процесса на 10 мс, которые в противном случае просто будут потеряны. Предоставить CPU возможность делать что-нибудь в то время, когда принтер переходит в состояние готовности, можно при помощи прерываний. Когда выполняется системный вызов печати строки, как мы уже показывали, буфер копируется в пространство ядра и первый символ строки копируется на принтер, как только принтер выставит бит готовности. После этого CPU вызывает планировщик, который запускает какой-либо другой процесс. Процесс, попросивший распечатать строку, оказывается заблокирован на весь период печати строки. Работа, выполняемая при системном вызове, показана на Рис. 7(а).

```
copy_from_user(buffer, p, count);
enable_interrupts();
while(*printer_status_reg != READY);
*printer_data_register = p[0];
scheduler();
```

А

```
if(count ==0)
{
        ublock_user();
}
else
{
       *printer_data_reg = p[i];
       count--;
       i++;
}
acknowledge_interrupt();
return_from_interrupt();
```

б

Рис. 7. Печать строки при помощи ввода-вывода, управляемого прерываниями: программа, выполняемая при обращении к системному вызову (а); процедура обработки прерываний (б)

Когда принтер напечатал символ и готов принять следующий, он инициирует прерывание. Это прерывание вызывает остановку текущего процесса и сохранение его состояния. Затем запускается процедура обработки прерывания от принтера. Грубый вариант этой программы показан на Рис. 7(б). Если напечатаны все символы,

обработчик прерывания предпринимает необходимые меры для разблокировки процесса пользователя. В противном случае он печатает следующий символ, подтверждает прерывание и возвращается к процессу, выполнение которого было приостановлено прерыванием от принтера.

Ввод-вывод с использованием DMA

Очевидный недостаток управляемого прерываниями ввода-вывода состоит в том, что прерывания происходят при печати каждого символа. Обработка прерываний занимает определенное время, поэтому такая схема не является эффективной. Решение этой проблемы заключается в использовании DMA. Идея состоит в том, чтобы позволить контроллеру DMA поставлять принтеру символы по одному, не беспокоя при этом CPU. По существу, этот метод почти не отличается от программного ввода-вывода, с той лишь разницей, что всю работу вместо CPU выполняет контроллер DMA. Набросок программы показан на Рис. 8.

```
copy_from_user(buffer, p, count);
setup_DMA_controller();
scheduler();
```

а

```
acknowledge_interrupt();
unblock_user();
return_from_interrupt();
```

б

Рис. 8. Печать строки при помощи DMA: программа, выполняемая при обращении к системному вызову (а); процедура обработки прерываний (б)

Наибольший выигрыш от использования DMA состоит в уменьшении количества прерываний с одного на печатаемый символ до одного на печатаемый буфер. Если символов много, а прерывания обрабатываются медленно, то этот выигрыш весьма существенен. С другой стороны, контроллер DMA обычно значительно уступает CPU в скорости. Если контроллер DMA не может— поддерживать полную скорость ввода или вывода с внешнего устройства, либо у CPU нет других задач во время ожидания прерывания от DMA, тогда оба предыдущих метода ввода-вывода (программный и управляемый прерываниями) будут предпочтительнее.

Программные уровни ввода-вывода

ПО ввода-вывода обычно организуется в виде четырех уровней, Рис. 9. У каждого уровня есть четко очерченная функция, которую он должен выполнять, и строго определенный интерфейс с соседними уровнями. Функции и интерфейсы уровней меняются от одной ОС к другой, поэтому последующее рассмотрение всех уровней, начиная с нижнего, не является специфичным для какой-либо конкретной машины.

Обработчики прерываний

Хотя программный ввод-вывод иногда бывает полезен, для большинства операций ввода-вывода прерывания являются неприятным, но необходимым фактом. Прерывания должны быть упрятаны как можно глубже во внутренностях ОС, чтобы о них знала как можно меньшая часть ОС. Лучший способ спрятать их заключается в блокировке драйвера, начавшего операцию ввода-вывода, вплоть до окончания этой операции и получения прерывания. Драйвер может заблокировать себя сам, выполнив на семафоре процедуру `down`, процедуру `wait` на переменной состояния, процедуру `receive` на сообщении, или что-либо подобное.

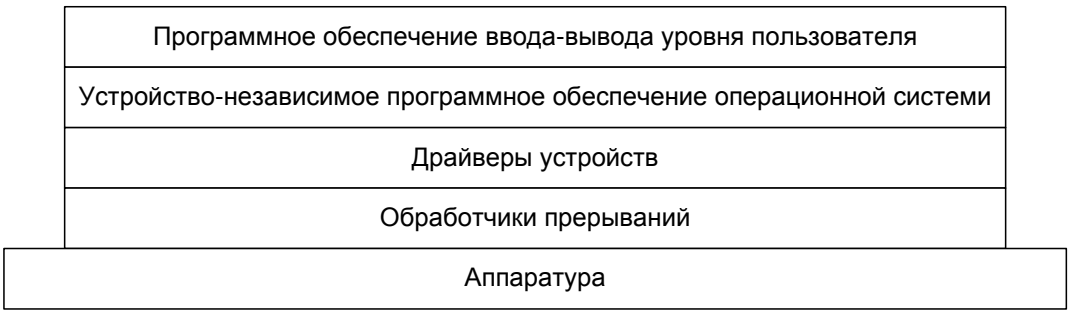

Рис. 9. Программные уровни ввода-вывода

Когда происходит прерывание, начинает работу обработчик прерываний. По окончании необходимой работы он может разблокировать драйвер, запустивший его. **В некоторых случаях** используется выполнение процедуры `up` на семафоре.

В других случаях обработчик прерываний вызывает процедуру монитора `signal` с переменной состояния. **В третьем случае** он посылает заблокированному драйверу сообщение. **В любом случае** драйвер разблокируется обработчиком прерываний. Эта схема лучше всего работает в драйверах, являющихся процессами ядра со своим собственным состоянием, стеком и счетчиком команд.

Конечно, в действительности все обстоит совсем не так просто. Обработать прерывание значительно сложнее, чем просто принять его, выполнить `up` на семафоре, после чего вернуться из прерывания в предыдущий процесс с помощью команды CPU IRET. ОС приходится выполнить значительно больше работы. Покажем схематичный набросок этой работы в виде набора шагов, которые следует выполнить ПО после того, как произошло аппаратное прерывание. Необходимо заметить, что детали во многом зависят от конкретной ОС, поэтому на каких-то машинах некоторые перечисленные шаги могут оказаться лишними, зато может потребоваться выполнение других, не помещенных в список шагов. Кроме того, на разных машинах может потребоваться выполнение перечисленных действий в разном порядке.

1. Сохранить все регистры (включая PSW), не сохраненные аппаратурой.

2. Установить контекст для процедуры обработки прерываний. Выполнение этого действия может включать установку TLB, MMU и таблицы страниц.

3. Установить указатель стека для процедуры обработки прерываний.

4. Выдать подтверждение контроллеру прерываний. Если централизованного контроллера прерываний нет, разрешить прерывания.

5. Скопировать содержимое регистров с того места, где они были сохранены (возможно, в каком-либо стеке), в таблицу процессов.

6. Запустить процедуру обработки прерываний. Она извлечет информацию из регистров контроллера устройства, инициировавшего прерывание.

7. Выбрать процесс, которому передать управление. Если прерывание разблокировало какой-либо высокоприоритетный процесс, он может быть выбран в качестве следующего.

8. Установить контекст MMU для следующего работающего процесса. Также может понадобиться определенная установка TLB.

9. Загрузить регистры нового процесса, включая его PSW.

10. Начать выполнение нового процесса.

Как можно заметить, обработка прерываний является далеко не простым делом. Она состоит из значительного количества команд CPU, особенно на машинах с виртуальной памятью, на которых необходимо восстанавливать состояние таблиц памяти или сохраненное состояние MMU (например, биты R и M). На некоторых машинах буфер быстрого преобразования адреса TLB и кэш CPU также требуют управления при переключении режимов пользователя и ядра, для чего необходимы дополнительные машинные циклы.

Драйверы устройств

Как было сказано, у каждого контроллера есть набор регистров, используемых для того, чтобы давать управляемому им устройству команды и читать состояние устройства. Число таких регистров и команды, выдаваемые устройствам, зависят от конкретного устройства. Например, драйвер мыши должен принимать от мыши информацию о том, насколько далеко она продвинулась по горизонтали и вертикали, а также о нажатых кнопках мыши. Драйвер диска, в отличие от драйвера мыши, должен знать о секторах, дорожках, цилиндрах, головках, их перемещении и времени установки, двигателях и тому подобных вещах, необходимых для правильной работы диска. Очевидно, что эти драйверы будут сильно различаться.

Поэтому для управления каждым устройством ввода-вывода, подключенным к компьютеру, требуется специальная программа. Эта программа, называемая

драйвером устройства, обычно пишется производителем устройства и распространяется вместе с устройством. Поскольку для каждой ОС требуются специальные драйверы, производители устройств обычно поставляют драйверы для нескольких наиболее популярных ОС.

Каждый драйвер устройства обычно поддерживает один тип устройств или, максимум, класс близких устройств. Например, драйвер SCSI-дисков обычно может поддерживать различные SCSI-диски, отличающиеся размерами и скоростями, и возможно даже будет поддерживать SCSI CD-ROM. С другой стороны, мышь и джойстик отличаются настолько сильно, что обычно требуют использования различных драйверов.

Чтобы получить доступ к аппаратной части устройства, то есть к регистрам контроллера, драйвер устройства должен быть частью ядра ОС, по крайней мере, в существующих на сегодняшний день архитектурах. В действительности, возможно создать и драйвер, работающий в пространстве пользователя, с системными вызовами для чтения и записи регистров устройств. В самом деле, это было бы даже неплохой идеей, так как позволило бы изолировать ядро от драйверов, а драйверы друг от друга. При этом была бы устранена основная причина крушения ОС — драйверы, содержащие ошибки, сталкивающиеся с ядром тем или иным образом. Тем не менее, поскольку современные ОС предполагают работу драйверов в ядре, рассмотрим здесь именно такую модель.

Так как в ОС будут устанавливаться куски программ (драйверы), написанные другими программистами, необходима определенная архитектура, позволяющая подобную установку. Это означает, что должна быть выработана строго определенная модель функций драйвера и его взаимодействия с остальной ОС. Драйверы устройств обычно располагаются под остальной ОС, как показано на Рис. 10.

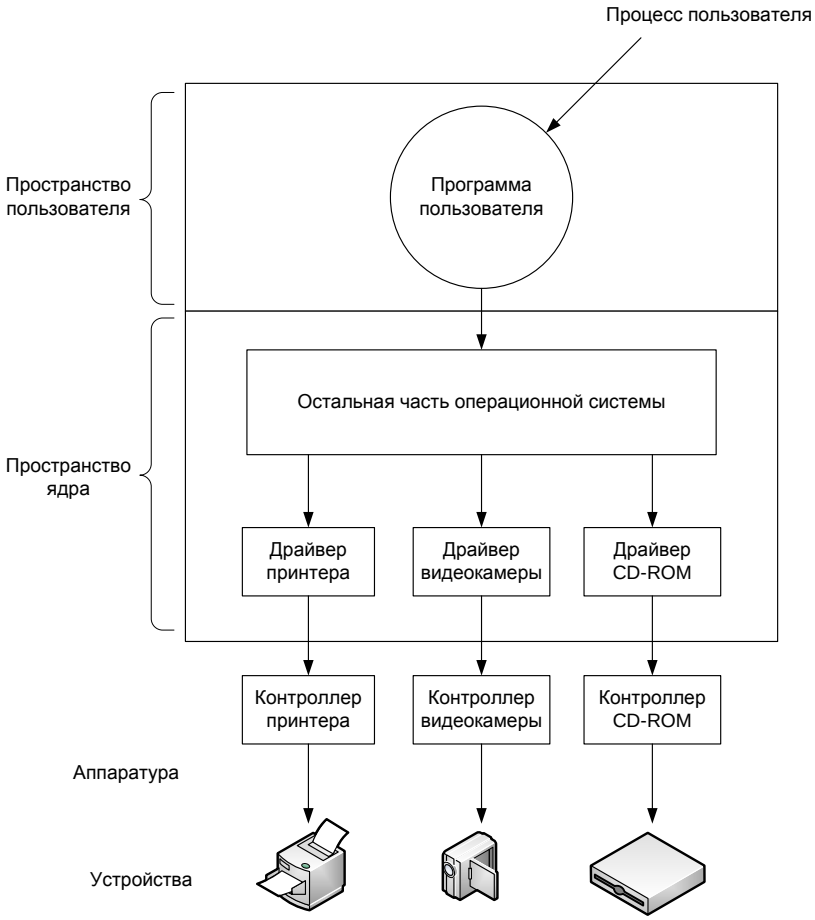

Рис. 10. Логическое расположение драйверов устройств. На самом деле весь обмен информацией между драйверами и контроллерами устройств идет по шине

ОС обычно классифицирует драйверы по нескольким категориям в соответствии с типами обслуживаемых ими устройств. К наиболее общим категориям относятся **блочные устройства**, например, диски, содержащие блоки данных, к которым возможна независимая адресация, и **символьные устройства**, такие как клавиатуры и принтеры, формирующие или принимающие поток символов.

В большинстве ОС определен стандартный интерфейс, который должны поддерживать все блочные драйверы, и второй стандартный интерфейс, поддерживаемый всеми символьными драйверами. Эти интерфейсы включают наборы процедур, которые могут вызываться остальной ОС для обращения к драйверу. К этим процедурам относятся, например, процедуры чтения блока (блочного устройства) или записи символьной строки (для символьного устройства).

В некоторых системах ОС представляет собой единую двоичную программу, содержащую в себе, в откомпилированном вместе с ней виде, все необходимые ей драйверы. Такая схема в течение многих лет была нормой для систем UNIX, так как они предназначались для работы в компьютерных центрах, а устройства ввода-вывода менялись нечасто. При добавлении нового устройства системный администратор просто перекомпилировал ядро с новым драйвером, получая новый двоичный модуль.

С появлением персональных компьютеров с их огромным разнообразием устройств ввода-вывода такая модель перестала работать. Далеко не все пользователи могли самостоятельно перекомпилировать и собрать ядро даже при наличии исходных текстов или объектных модулей, что, кстати, также не всегда имеет место. Вместо этого ОС, начиная с MS-DOS, перешли к модели динамической подгрузки драйверов во время выполнения системы. Различные системы загружают драйверы по-разному.

У драйвера устройства есть несколько функций. Наиболее **очевидная функция драйвера состоит в обработке абстрактных запросов чтения и записи независимого от устройств ПО, расположенного над ними**. Но кроме этого они должны также выполнять еще несколько функций. Например, драйвер **должен при необходимости инициализировать устройство**. Ему также может понадобиться **управлять энергопотреблением устройства и регистрацией событий**.

Многие драйверы устройств обладают сходной общей структурой. Типичный драйвер **начинает с проверки входных параметров**. Если они не удовлетворяют определенным критериям, драйвер возвращает ошибку. В противном случае **драйвер преобразует абстрактные термины в конкретные**. Например, дисковый драйвер может преобразовывать линейный номер блока в номера головки, дорожки и секторы.

Затем драйвер может **проверить, не используется ли это устройство в данный момент**. Если устройство занято, запрос может быть поставлен в очередь. Если устройство свободно, **проверяется аппаратный статус устройства**, чтобы понять, может ли запрос быть обслужен прямо сейчас. Может оказаться необходимым **включить устройство или запустить двигатель**, прежде чем **начнется перенос данных**. Как только устройство включено и готово, может начинаться собственно **управление устройством**.

Управление устройством подразумевает выдачу ему серии команд. Именно в драйвере определяется последовательность команд в зависимости от того, что должно быть сделано. Определившись с командами, драйвер начинает **записывать их в регистры контроллера устройства**. После записи каждой команды в контроллер может быть нужно **проверить, принял ли контроллер эту команду и готов ли принять следующую**. Такая последовательность действий продолжается до тех пор, пока контроллеру не будут даны все команды. Некоторые контроллеры способны принимать связные списки команд, находящихся в RAM. Они сами считывают и выполняют их без дальнейшей помощи ОС.

После того как драйвер передал все команды контроллеру, ситуация может развиваться по двум сценариям. Во многих случаях **драйвер устройства должен ждать**, пока контроллер не выполнит для него определенную работу, поэтому он блокируется до тех пор, пока прерывание от устройства его не разблокирует. В других случаях операция **завершается без задержек и драйверу не нужно блокироваться**. Например, для скроллинга экрана в символьном режиме нужно записать всего лишь несколько байтов в регистры контроллера. Вся операция занимает несколько нс.

В первом случае заблокированный драйвер будет активизирован прерыванием. **Во втором случае драйвер не блокируется**. В любом случае по завершении выполнения операции драйвер должен проверить, завершилась ли операция без ошибок. Если все в порядке, драйверу, возможно, придется предать данные (например, только что прочитанный блок) независимому от устройств ПО. Наконец, драйвер возвращает некоторую информацию о состоянии для информирования вызывающей программы о статусе завершения операции. Если в очереди находились другие запросы, один из них теперь может быть выбран и запущен. В противном случае драйвер блокируется в ожидании следующего запроса.

Эта упрощенная модель является лишь грубым приближением к реальности. На самом деле программа значительно сложнее, причиной чему служит большое количество разнообразных факторов:

- Во-первых, устройство ввода-вывода может завершить выполнение операции во время работы драйвера, таким образом прерывая его работу.
- Во-вторых, во время обработки сетевым драйвером пришедшего пакета может прийти еще один пакет. Соответственно, драйвер должен быть **реентерабельным**, то есть должен быть готов к тому, что во время обработки первого вызова может последовать другой вызов.

В системе с возможностью PnP установки устройства могут добавляться или удаляться во время работы ОС. В результате в то время, когда драйвер занят чтением с какого-либо устройства, система может проинформировать его, что пользователь внезапно удалил это устройство из системы. При этом не только текущая операция переноса данных должна быть прервана без повреждения структур данных ядра, но также и все ожидающие обработки запросы к теперь исчезнувшему устройству должны быть корректно удалены из ОС, а обратившимся к ним программам должна быть сообщена неприятная новость. Более того, неожиданное добавление нового устройства может заставить ядро жонглировать ресурсами (например, линиями запросов прерывания), отнимая их у одного драйвера и предоставляя другому драйверу.

Драйверам не разрешается обращаться к системным вызовам, но им часто бывает необходимо взаимодействовать с остальным ядром. Обычно разрешаются обращения к некоторым системным процедурам. Например, драйверы обращаются к системным процедурам для выделения им аппаратно фиксированных страниц памяти в качестве буферов, а также затем, чтобы вернуть эти страницы обратно ядру. Кроме того, драйверы пользуются вызовами, управляющими MMU, таймерами, контроллером DMA, контроллером прерываний и т. п.

Независимое от устройств ПО ввода-вывода

Хотя некоторая часть ПО ввода-вывода предназначена для работы с конкретными устройствами, другая часть является независимой от устройств. Расположение точной границы между драйверами и независимым от устройств ПО зависит от системы (и устройств), так как некоторые функции, которые могут быть выполнены независимым от устройств образом, часто выполняются прямо в драйверах из различных соображений, в том числе соображений эффективности. Функции, показанные в табл. 1, обычно реализуются в независимом от устройств ПО.

Таблица 1. Функции независимого от устройств ПО

Единообразный интерфейс для драйверов устройств
Буферизация
Сообщение об ошибках
Захват и освобождение выделенных устройств
Размер блока, не зависящий от устройства

Основная задача независимого от устройств ПО состоит в выполнении функций ввода-вывода, общих для всех устройств, и предоставлении единообразного интерфейса для программ уровня пользователя.

Единообразный интерфейс для драйверов устройств

Главный вопрос ОС — как сделать так, чтобы все устройства ввода-вывода и драйверы выглядели более-менее одинаково. Если диски, принтеры, клавиатуры и т. д. требуют различных интерфейсов, при появлении каждого нового устройства будет требоваться переделка ОС, что определенно не является удачной мыслью.

Этот вопрос связан с интерфейсом между драйверами устройств и остальной ОС. На Рис. 11(а) показана ситуация, в которой у всех драйверов различный интерфейс с ОС. Это означает, что функции драйвера, доступные системе, отличаются от драйвера к драйверу. Это может также означать, что функции ядра, необходимые для драйвера, тоже различаются. Все вместе это означает, что взаимодействие с каждым новым драйвером требует больших усилий программистов.

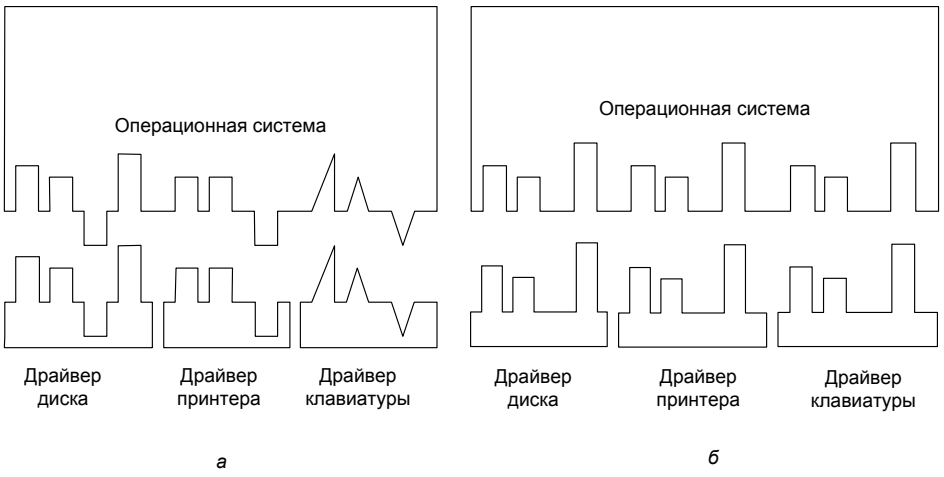

Рис. 11. Стандартный интерфейс драйверов отсутствует (а); стандартный интерфейс драйверов присутствует (б)

Напротив, на Рис. 11(б) изображен принципиально другой подход, при котором у всех драйверов один и тот же интерфейс. **При этом значительно легче установить новый драйвер, при условии, что он соответствует стандартному интерфейсу**. Это также означает, что программисты, пишущие драйверы, знают, чего от них ожидают (то есть какие функции они должны реализовать и к каким функциям ядра они могут обращаться). На практике не все устройства являются абсолютно идентичными, но обычно имеется небольшое число типов устройств, достаточно сходных друг с другом. Например, даже у блочных и символьных устройств есть много общих функций.

Другой аспект единообразного интерфейса состоит в именовании устройств ввода-вывода. Независимое от устройств ПО занимается отображением символьных имен устройств на соответствующие драйверы. Например, в системе UNIX имя устройства /dev/disk0 однозначно указывает i-узел специального файла, содержащий **номер главного устройства**, использующийся для определения расположения подходящего драйвера. Этот i-узел также содержит **номер второстепенного устройства**, передаваемый в виде параметра драйверу для указания конкретного диска или раздела диска, к которому относится операция чтения или записи. Все устройства в системе UNIX имеют главный и второстепенный номера, по которым они однозначно идентифицируются. Выбор всех драйверов осуществляется по главному номеру устройства.

С именованием устройств тесно связан вопрос защиты. Как ОС предотвращает доступ пользователей к устройствам, на который у них нет прав? В UNIX и в Windows based устройства представляются в файловой системе в виде именованных объектов, что дает возможность применять обычные правила защиты файлов к устройствам ввода-вывода. Таким образом, системный администратор может установить нужные разрешения для каждого устройства.

Буферизация

Буферизация также является важным вопросом как для блочных, так и для символьных устройств по самым разным причинам. Чтобы проиллюстрировать одну из причин, рассмотрим процесс, который хочет прочитать данные с модема. Одна из возможных стратегий обработки поступающих символов состоит в обращении процесса пользователя к системному вызову `read` и блокировке в ожидании отдельного символа. Каждый прибывающий символ вызывает прерывание. Процедура обработки прерываний передает символ пользовательскому процессу и разблокирует его. Поместив куда-либо полученный символ, процесс читает следующий символ, опять блокируясь. Эта схема проиллюстрирована на Рис. 12(а).

Недостаток такого подхода состоит в том, что процесс пользователя должен быть активирован при прибытии каждого символа, что крайне неэффективно.

Улучшенный вариант показан на Рис. 12(б). Здесь пользовательский процесс предоставляет буфер размером в n символов в пространстве пользователя, после чего выполняет чтение n символов. **Процедура обработки прерываний помещает приходящие символы в буфер до тех пор, пока он не заполнится.** Затем она

активизирует процесс пользователя. Такая схема гораздо эффективнее предыдущей, однако у нее также есть недостатки: что случится, если в момент прибытия символа страница памяти, в которой расположен буфер, окажется выгруженной из физической памяти? Конечно, буфер можно зафиксировать в памяти; если слишком много процессов начнут фиксировать свои страницы в памяти, пул доступных страниц памяти уменьшится, в результате чего снизится производительность.

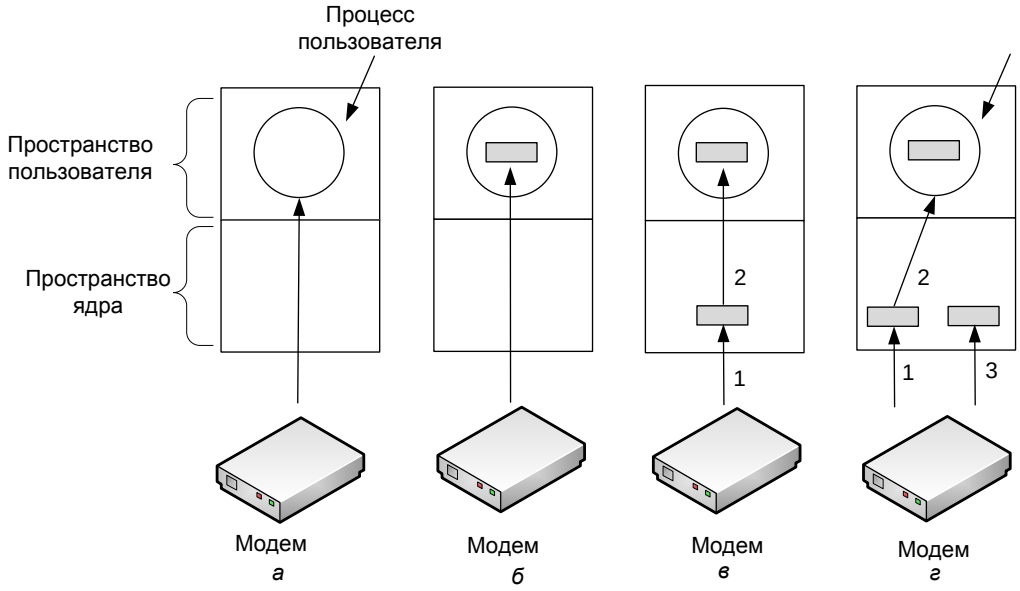

Рис. 12. Небуферизированный ввод (а); буферизация в пространстве пользователя (б); буферизация в ядре с копированием в пространство пользователя (в); двойная буферизация в ядре (г)

Следующий подход состоит в создании буфера, в который обработчик прерываний будет помещать поступающие символы, в ядре, как показано на Рис. 12(в). Когда этот буфер наполняется, достается страница с буфером пользователя, и содержимое буфера копируется туда за одну операцию. Такая схема намного эффективнее.

Однако даже при использовании этой схемы имеются определенные проблемы. Что случится с символами, прибывающими в тот момент, когда страница пользователя загружается с диска? Поскольку буфер полон, их некуда поместить. Решение проблемы состоит в использовании второго буфера в ядре, в который помещаются символы после заполнения первого буфера до его освобождения, Рис. 12(г). При этом буферы как бы меняются местами, то есть первый буфер начинает играть роль запасного. Такая схема часто называется **двойной буферизацией**.

Буферизация также играет важную роль при выводе данных. Рассмотрим, например, как происходит вывод на модем при помощи модели, показанной на Рис. 12(б). Пользовательский процесс выполняет системный вызов `write` для вывода n символов. У системы в этот момент есть выбор. Она может заблокировать пользователя, пока все символы не будут записаны, но по медленной телефонной линии это может занять довольно много времени. Она может разрешить пользователю продолжать выполнение немедленно и выполнять операцию ввода-вывода, в то время как пользователь занимается другими вычислениями. Однако при этом возникает непростая проблема: как пользовательский процесс узнает, что операция вывода завершена и он может опять пользоваться этим буфером? Система может подать сигнал или программное прерывание, но такой стиль программирования сложен и чреват возникновениями ситуаций состязания. Значительно лучшее решение состоит в копировании данных в буфер ядра, по аналогии с Рис. 12(в) (но в обратном направлении), и немедленном разблокировании вызывающего процесса. Теперь уже не важно, когда будет выполнена физическая операция ввода-вывода. Пользователь может использовать буфер сразу после возврата ему управления.

Буферизация является широко применяемой технологией, однако у нее имеются свои недостатки. **Если при буферизации данные копируются слишком много раз, страдает производительность**. Рассмотрим, например, сеть (Рис. 13). Пользователь

обращается к системному вызову для записи данных по сети. Ядро копирует пакет в буфер ядра, чтобы пользователь мог немедленно продолжить работу (шаг 1).

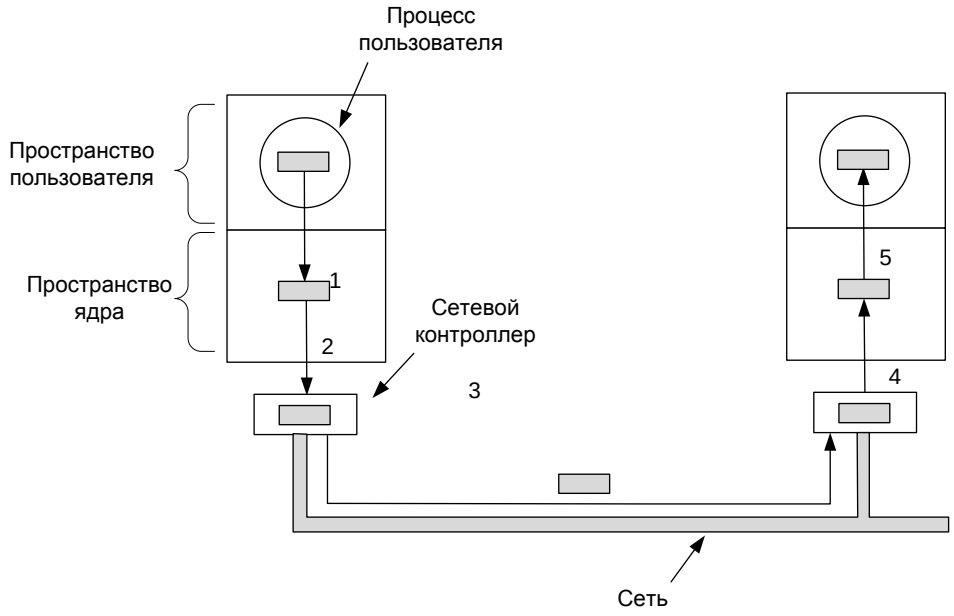

Рис. 13. Копии пакета при передаче его по сети

При вызове драйвера пакет копируется в контроллер для вывода (шаг 2). Память ядра не используется напрямую, так как передача по линии должна производиться с постоянной скоростью. Драйвер не может гарантировать, что он сможет получать доступ к памяти с постоянной скоростью, так как каналы DMA и другие устройства ввода-вывода могут внезапно захватить много циклов шины. Если драйвер не сможет предоставить контроллеру вовремя хотя бы одно слово данных, весь пакет будет разрушен. Этой проблемы удается избежать при помощи буферизации пакета целиком внутри контроллера.

Затем пакет копируется в сеть (шаг 3). Получающая сторона собирает пакет из отдельных битов также в буфере сетевого контроллера. После этого пакет копируется в буфер ядра получателя (шаг 4). Наконец, он копируется в буфер получающего процесса (шаг 5). Обычно в ответ получатель отправляет подтверждение. Получив подтверждение, отправитель может посылать следующий пакет. Однако должно быть очевидно, что все эти операции копирования существенно замедляют передачу данных, поскольку все эти шаги должны выполняться последовательно.

Сообщения об ошибках

Ошибки значительно чаще случаются в контексте ввода-вывода, нежели в других контекстах. При возникновении ошибок ОС должна обработать их как можно быстрее. Многие ошибки являются специфичными для конкретного устройства и должны обрабатываться соответствующим драйвером, но структура обработки ошибок является независимой от устройств.

Один из классов ошибок ввода-вывода составляют ошибки программирования. Они происходят, когда процесс просит чего-либо невозможного — например, записать данные на устройство ввода (клавиатуру, мышь, сканер и т. д.) или, наоборот, прочитать данные с устройства вывода (принтера, плоттера и т. п.). Другие ошибки включают предоставление неверного адреса буфера или иных параметров, а также обращение к несуществующему устройству (например, к диску 3, когда у системы всего два диска). Обработка таких ошибок довольно проста: код ошибки возвращается вызывающему процессу.

К другому классу ошибок относятся собственно ошибки ввода-вывода, такие как попытка записать в поврежденный блок диска или попытка прочитать данные с выключенной видеокамеры. В таких случаях драйвер сам решает, что ему делать. Если драйвер не может сам принять решение, он передает сведения о возникшей проблеме в вышестоящие инстанции (независимому от устройств ПО).

Действия этого ПО зависят от окружения и природы ошибки. Если это простая ошибка чтения, а программа интерактивная, можно отобразить диалоговое окно с вопросом к пользователю о дальнейших действиях. В качестве выбора может быть предложено повторение попытки определенное число раз, игнорирование ошибки или уничтожение процесса. Если программа не интерактивная, единственная возможность состоит в аварийном завершении системного вызова с соответствующим кодом ошибки.

Однако не все ошибки могут быть обработаны подобным образом. Например, может оказаться поврежденной критическая структура данных, такая как корневой каталог или список свободных блоков. В этом случае, возможно, ОС придется вывести сообщение об ошибке и завершить свою работу.

Захват и освобождение выделенных устройств

Некоторые устройства, например устройство записи компакт-дисков, могут использоваться в **каждый момент времени только одним пользователем**. ОС должна рассмотреть запросы на использование этого устройства и либо принять их, либо отказать в выполнении запроса, в зависимости от доступности запрашиваемого устройства. Простой способ обработки этих запросов состоит в требовании к процессам обращаться напрямую к системному вызову `open` по отношению к специальным файлам для данных устройств. Если устройство недоступно, системный вызов `open` завершится неуспешно. Обращение к системному вызову `close` освобождает устройство.

Альтернативный подход состоит в предоставлении специального механизма для запроса и освобождения выделенных устройств. Попытка захватить недоступное устройство вызовет блокировку вызывающего процесса вместо возврата с ошибкой. Блокированные процессы устанавливаются в очередь. Раньше или позже запрашиваемое устройство освобождается и первому процессу в очереди разрешается захватить его и продолжить выполнение.

Независимый от устройств размер блока

У различных дисков могут быть разные размеры сектора. Независимое от устройств ПО должно скрывать этот факт от верхних уровней и предоставлять им единообразный размер блока, например, объединяя несколько физических сегментов в один логический блок. При этом более высокие уровни имеют дело только с абстрактными устройствами, с одним и тем же размером логического блока, не зависящим от размера физического сектора. Некоторые символьные устройства предоставляют свои данные побайтно (например, модемы), тогда как другие выдают их большими порциями (например, сетевые интерфейсы). Эти различия также могут быть скрыты.

ПО ввода-вывода пространства пользователя

Хотя большая часть ПО ввода-вывода находится в ОС, небольшие его порции состоят из **библиотек**, присоединенных к программам пользователя, или даже целых программ, работающих вне ядра. Системные вызовы, включая системные вызовы ввода-вывода, обычно состоят из библиотечных процедур. Если программа на С содержит вызов

```
count = write(fd, buffer, nbytes);
```

библиотечная процедура `write` будет скомпонована с программой и, таким образом, будет содержаться в двоичной программе, загружаемой в память во время выполнения программы. Набор всех этих библиотечных процедур, несомненно, является частью системы ввода-вывода.

Хотя многие такие процедуры мало что делают, кроме обращения к системному вызову с соответствующими параметрами, есть ряд процедур ввода-вывода, выполняющих определенную работу. В частности, библиотечными процедурами выполняются операции форматного ввода и вывода. Например, процедура `printf` языка С, принимающая на входе текстовую строку и, возможно, несколько переменных, создает из нее ASCII-строку, после чего обращается к системному вызову `write` для вывода строки. Для примера рассмотрим следующий оператор:

```
printf("Квадрат числа %3d равен %6d\n", i, i*i);
```

Он форматирует строку, состоящую из 14-символьной строки "Квадрат числа", за которой следует значение переменной i в виде 3-символьной строки, затем 7-символьной строки "равен", потом i^2 в виде 6 символов и, наконец, символа перевода строки.

Примером сходной процедуры ввода может служить `scanf`, читающая текстовую строку и преобразующая ее в значения переменных в соответствии с форматом, сходным с используемым процедурой `printf`. Стандартная библиотека ввода-вывода содержит большое количество процедур, включающих операции ввода-вывода и работающих как часть программы пользователя.

Не все ПО ввода-вывода пространства пользователя состоит из библиотечных процедур. Другую важную категорию составляет система спулинга. **Спулинг** (spooling — подкачка, предварительное накопление данных) представляет собой способ работы с выделенными устройствами в многозадачной системе. Рассмотрим типичное устройство, на котором используется спулинг - принтер. В принципе можно разрешить каждому пользователю открывать специальный символьный файл принтера, однако представьте себе, что процесс открыл его, а затем не обращался к принтеру в течение нескольких часов. Ни один другой процесс в это время не сможет ничего напечатать.

Вместо этого создается специальный процесс, называемый **демоном**, и специальный каталог, называемый **каталогом спулинга или каталогом спулера**. Чтобы распечатать файл, процесс сначала создает специальный файл, предназначенный для печати, который помещает в каталог спулинга. Этот файл печатает демон, единственный процесс, которому разрешается пользоваться специальным файлом принтера. Таким образом, потенциальная проблема, связанная с тем, что какой-либо процесс на слишком долгий срок захватит принтер, решается при помощи защиты специального файла принтера от прямого доступа пользователей.

Спулинг используется не только для принтеров. Например, передачу файлов по сети также часто осуществляет специальный сетевой демон. Чтобы послать куда-либо файл, пользователь помещает его в каталог сетевого демона. Затем сетевой демон извлекает оттуда файл и посылает по сети.

На Рис. 14 показана структура системы ввода-вывода, со всеми уровнями и основными функциями каждого уровня. Снизу вверх эти уровни представляют собой аппаратуру, обработчики прерываний, независимое от устройств ПО и, наконец, процессы пользователя.

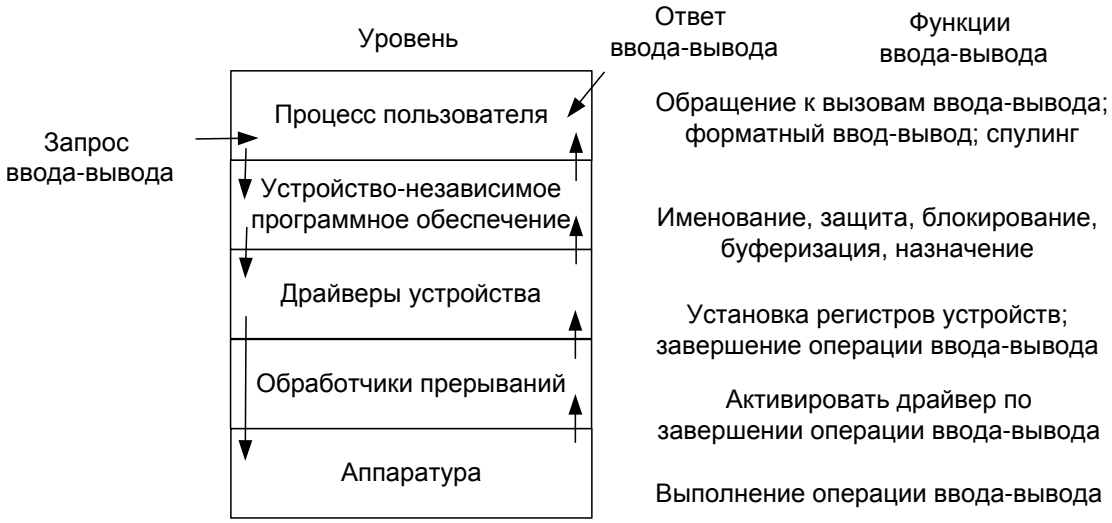

Рис. 14. Уровни и основные функции системы ввода-вывода

Стрелки на Рис. 14 изображают поток управления. Например, когда программа пользователя пытается прочитать блок из файла, для обработки вызова запускается ОС. Независимое от устройств ПО ищет этот блок в кэше. Если требуемого блока там нет, оно вызывает драйвер устройства, чтобы обратиться к аппаратуре и получить этот блок с диска. При этом процесс блокируется до тех пор, пока не завершится дисковая операция.

Когда диск завершает операцию, аппаратура инициирует прерывание. Обработчик прерываний запускается, чтобы определить, что случилось, то есть какое устройство требует внимания. Затем он извлекает статус устройства и активизирует «спящий» процесс, чтобы завершить запрос ввода-вывода и предоставить пользовательскому процессу возможность продолжать.

Литература

1. Э. Таненбаум. Современные операционные системы. 2-ое изд. –СПб.: Питер, 2002. – 1040 с.

2. А. Шоу. Логическое проектирование операционных систем. Пер. с англ. –М.: Мир, 1981. –360 с.

3. С. Кейслер. Проектирование операционных систем для малых ЭВМ: Пер. с англ. –М.: Мир, 1986. –680 с.

4. Э. Таненбаум, А. Вудхалл. Операционные системы: разработка и реализация. Классика CS. –СПб.: Питер, 2006. –576 с.

5. Microsoft Development Network. URL: http://msdn.com