* + 1. **ИС2К Операционные системы**

**Разработка систем страничной организации памяти**

1. **Сравнительный анализ локальной и глобальной политики**

В предыдущих разделах мы рассмотрели несколько алгоритмов выбора удаляемой страницы в случае возникновения ошибки отсутствия страницы.

Главный вопрос, связанный с этим выбором (который мы до сих пор тщательно обходили стороной): как должна быть распределена память между конкурирующими работоспособными процессами?

Посмотрите на рис. 3.20, *а.* На нем изображены три процесса: А, *В* и С, составляющие набор работоспособных процессов. Предположим, что процесс *А* сталкивается с ошибкой отсутствия страницы. Должен ли алгоритм замещения страниц попытаться найти наиболее давно использованную страницу, рассматривая лишь шесть страниц, выделенных на данный момент процессу *А*, или же он должен рассматривать все страницы, имеющиеся в памяти? Если он осуществляет поиск только среди страниц, принадлежащих процессу А, то страницей с наименьшим значением возраста будет А5, и мы получим ситуацию, показанную на рис. 3.20, *б.*

**Возраст**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **АО** | **10** | **АО** |  | **АО** |
| **А1** | **7** | **А1** |  | **А1** |
| **А2** | **5** | **А2** |  | **А2** |
| **АЗ** | **4** | **АЗ** |  | **АЗ** |
| **А4** | **6** | **А4** |  | **А4** |
| **А5** | **3** | ***<Ж>*** |  | **А5** |
| **ВО** | **9** | **ВО** |  | **ВО** |
| **В1** | **4** | **В1** |  | **В1** |
| **В2** | **6** | **В2** |  | **В2** |
| **ВЗ** | **2** | **ВЗ** |  | ***<Ж>*** |
| **В4** | **5** | **В4** |  | **В4** |
| **В5** | **6** | **В5** |  | **В5** |
| **В6** | **12** | **В6** |  | **В6** |
| **С1** | **3** | **С1** |  | **С1** |
| **С2** | **5** | **С2** |  | **С2** |
| **СЗ** | **6** | **СЗ** |  | **СЗ** |

*а* ***б в***

Рис. 3.20. Сравнение локального и глобального алгоритмов замещения страниц:
*а* — исходная конфигурация; *б* — работа локального алгоритма замещения страниц;
*в* — работа глобального алгоритма замещения страниц

В то же время, если страница с наименьшим значением возраста удаляется независимо от того, какому процессу она принадлежит, то будет выбрана страница *В3,* и мы получим ситуацию, показанную на рис. 3.20, *в.* Алгоритм, чья работа показана на рис. 3.20, б, называется **локальным** алгоритмом замещения страниц, а алгоритм, чья работа показана на рис. 3.20, *в,* называется **глобальным** алгоритмом замещения страниц. Локальный алгоритм хорошо подходит для выделения каждому процессу фиксированной доли памяти. При использовании глобальных алгоритмов страничные блоки распределяются среди работоспособных процессов в динамическом режиме. Поэтому со временем изменяется количество страничных блоков, выделенных каждому процессу.

В целом глобальные алгоритмы работают лучше, особенно когда размер рабочего набора может изменяться в течение жизненного цикла процесса. Если используется локальный алгоритм, а рабочий набор разрастается, то это приводит к пробуксовке даже при избытке свободных страничных блоков. Если рабочий набор сужается, локальные алгоритмы приводят к нерациональному использованию памяти. Если используется глобальный алгоритм, система должна постоянно принимать решение о том, сколько страничных блоков выделить каждому процессу. Одним из способов может стать отслеживание размера рабочего набора на основе показаний битов возраста, но не факт, что этот подход предотвратит пробуксовку. Рабочий набор может изменять свой размер за миллисекунды, в то время как весьма приблизительные показатели на основе битов возраста складываются за несколько тактов системных часов.

Другой подход заключается в использовании алгоритма выделения процессам страничных блоков. Один из способов заключается в периодическом определении количества работающих процессов и выделении каждому процессу равной доли. Таким образом, при наличии 12 416 доступных (то есть не принадлежащих операционной системе) страничных блоков и 10 процессов каждый процесс получает 1241 страничный блок. Оставшиеся шесть блоков переходят в резерв для использования в случае возникновения ошибки отсутствия страницы.

Хотя этот метод может показаться справедливым, едва ли есть смысл предоставлять одинаковые доли памяти процессу в 10 Кбайт и процессу в 300 Кбайт. Вместо этого страницы должны быть распределены пропорционально общему размеру каждого процесса, чтобы процессу в 300 Кбайт было выделено в 30 раз больше блоков, чем процессу в 10 Кбайт. Наверное, разумно будет дать каждому процессу какое-то минимальное количество, чтобы он мог запуститься независимо от того, насколько малым он будет. К примеру, на некоторых машинах одиночная двухоперандная команда может нуждаться не менее чем в шести страницах, потому что на границах страниц может оказаться все: сама команда, операнд-источник и операнд-приемник. При выделении всего лишь пяти страниц программа, содержащая подобные команды, вообще не сможет выполняться.

При использовании глобального алгоритма может появиться возможность запускать каждый процесс с некоторым количеством страниц пропорционально размеру процесса, но как только процессы будут запущены, распределение должно динамически обновляться. Одним из способов управления распределением является использование алгоритма **частоты возникновения ошибки отсутствия страницы** (Page Fault Frequency (**PFF**)). Он подсказывает, когда нужно увеличивать или уменьшать количество выделенных процессу страниц, но ничего не говорит о том, какую страницу следует удалить в случае возникновения ошибки. Он всего лишь контролирует размер выделенного набора.

Ранее уже говорилось, что для большого класса алгоритмов замещения страниц, включая LRU, известно, что чем больше выделено страниц, тем меньше уровень ошибок. Данное предположение положено в основу алгоритма PFF. Это свойство проиллюстрировано на рис. 3.21.

Измерение уровня ошибок отсутствия страниц осуществляется простым подсчетом количества ошибок в секунду, можно также взять скользящее среднее за несколько прошедших секунд. Одним из простых методов осуществления подобного измерения является прибавление количества ошибок отсутствия страниц в течение только что прошедшей секунды к текущему значению скользящего среднего и деление результата на два. Пунктирная линия, обозначенная буквой A, соответствует неприемлемо высокому

Рис. 3.21. Уровень ошибок как функция от количества выделенных
страничных блоков

уровню ошибок отсутствия страницы, поэтому, чтобы снизить этот уровень, процесс, в котором происходят ошибки, получает больше страничных блоков. Пунктирная линия, обозначенная буквой B, соответствует слишком низкому уровню ошибок отсутствия страницы, который позволяет предположить, что процессу выделено чрезмерно много памяти. В этом случае страничные блоки у него могут быть отобраны. Таким образом алгоритм PFF пытается сохранить для каждого процесса уровень подкачки страниц в пределах приемлемых границ.

Важно отметить, что некоторые алгоритмы замещения страниц могут работать как с локальной, так и с глобальной политикой замещения. Например, FIFO может заменять самые старые страницы во всем пространстве памяти (глобальный алгоритм) или самые старые страницы, принадлежащие текущему процессу (локальный алгоритм). Точно так же алгоритм LRU или приближения к его реализации могут заменять наиболее давно использованную страницу во всей памяти (глобальный алгоритм) или наиболее давно использованную страницу, принадлежащую текущему процессу (локальный алгоритм). В некоторых случаях выбор локальной, а не глобальной стратегии не зависит от алгоритма.

В то же время для других алгоритмов замещения страниц имеет смысл только локальная стратегия. В частности, алгоритмы рабочего набора и WSClock обращаются к некоторым конкретным процессам и должны применяться в этом контексте. По сути, не существует рабочего набора для всей машины, и при попытке воспользоваться объединением всех рабочих наборов алгоритм утратит свойство локальности и не сможет работать эффективно.

1. **Управление загрузкой**

Даже при самом лучшем алгоритме замещения страниц и оптимальной системе распределения страничных блоков между процессами система может пробуксовывать. Фактически когда сумма рабочих наборов всех процессов превышает объем памяти, можно ожидать пробуксовки. Одним из симптомов такой ситуации является показание алгоритма PFF, свидетельствующее о том, что некоторые процессы нуждаются в дополнительной памяти, но ни одному из процессов не нужен ее меньший объем. В таком случае нуждающимся процессам невозможно предоставить дополнительную память, не «ущемляя» другие процессы. Единственным реальным решением станет избавление от некоторых процессов.

Неплохим способом сокращения количества соревнующихся за обладание памятью процессов является сброс некоторых из них на диск и освобождение всех удерживавшихся ими страниц. Например, один процесс может быть сброшен на диск, а его страничные блоки — поделены между другими пробуксовывающими процессами. Если пробуксовка прекратится, то система некоторое время может проработать в таком состоянии. Если она не прекратится, потребуется сбросить на диск другой процесс и продолжать в том же духе, пока пробуксовка не прекратится. Таким образом, даже при страничной организации памяти свопинг может не утратить своей актуальности, только теперь он используется для сокращения потенциальной потребности в памяти, а не для повторного востребования страниц.

Свопинг процессов, призванный снизить нагрузку на память, напоминает двухуровневую диспетчеризацию, при которой часть процессов помещается на диск, а для распределения оставшихся процессов используется краткосрочный диспетчер. Понятно, что эти две идеи можно сочетать, выгружая достаточное количество процессов, чтобы достичь приемлемого уровня ошибок отсутствия страницы. Периодически какие-то процессы загружаются с диска, а какие-то выгружаются на него.

Еще одним фактором, требующим рассмотрения, является степень многозадачности. Когда в основной памяти находится слишком мало процессов, центральный процессор может весьма существенные периоды времени быть недогружен. С учетом этого при принятии решения о выгрузке процесса нужно принимать во внимание не только его размер и уровень подкачки страниц, но и его характеристики, например ограничен ли этот процесс скоростью вычислений или же он ограничен скоростью работы устройств ввода-вывода, и принимать во внимание те характеристики, которыми обладают остальные процессы.

1. **Размер страницы**

Размер страницы является тем параметром, который должен быть выбран операционной системой. Даже если аппаратура была разработана, к примеру, под страницы в 4096 байт, операционная система может запросто рассматривать пары страниц 0 и 1, 2 и 3, 4 и 5 и т. д. как страницы размером 8 Кбайт, всегда выделяя им два последовательных страничных блока размером 8192 байт.

Определение наилучшего размера страницы требует сохранения баланса между несколькими конкурирующими факторами. Таким образом, абсолютно оптимального решения не существует. Для начала возьмем два фактора, являющихся аргументами в пользу небольшого размера страницы. Произвольно взятые текст, данные или сегмент стека не заполнят целое число страниц. В среднем половина последней страницы останется незаполненной. Оставшееся пространство на этой странице тратится впустую. Эти потери называются **внутренней фрагментацией**. Если в памяти *п* сегментов, а размер страницы составляет *р* байт, то на внутреннюю фрагментацию будет потрачено впустую пр/2 байт. Эти соображения являются аргументом в пользу небольшого размера страницы.

Другой аргумент в пользу небольшого размера страницы возникает при рассмотрении программы, состоящей из восьми последовательных этапов, каждый размером 4 Кбайт. При размере страницы 32 Кбайт программе всегда должно быть выделено 32 Кбайт. При странице размером 16 Кбайт ей потребуется только 16 Кбайт. А при размере страницы 4 Кбайт или меньше ей в любой момент времени потребуется только 4 Кбайт.

В общем, при большом размере страницы в памяти будет больше неиспользуемого пространства, чем при малом.

В то же время при небольших по объему страницах подразумевается, что программы будут нуждаться в большом количестве страниц, а это приведет к большому размеру таблицы страниц. Программе размером 32 Кбайт требуется только 4 страницы по 8 Кбайт, но 64 страницы по 512 байт. Как правило, на диск и с диска переносится сразу вся страница, при этом основная часть времени тратится на задержки, связанные с поиском нужного сектора и раскруткой диска, поэтому перенос небольшой страницы занимает практически столько же времени, что и перенос большой страницы. Для загрузки 64 страниц по 512 байт может потребоваться 64 • 10 мс, а для загрузки четырех страниц по 8 Кбайт — всего 4 • 12 мс.

Кроме того, страницы небольшого размера расходуют много ценного пространства в TLB. Предположим, что ваша программа использует 1 Мбайт памяти с рабочим набором размером 64 Кбайт. При страницах размером 4 Кбайт программа будет занимать как минимум 16 записей в TLB. При страницах размером 2 Мбайт было бы достаточно одной записи в TLB (теоретически вам может потребоваться разделить данные и инструкции). Из-за дефицита TLB-записей и их значимого влияния на производительность приходится везде, где только можно, расплачиваться использованием больших страниц. Чтобы сбалансировать все эти компромиссы, операционные системы иногда используют разные размеры страниц для разных частей системы. Например, большие страницы для ядра и меньшие по размеру страницы для пользовательских процессов.

На некоторых машинах таблица страниц должна быть загружена (операционной системой) в аппаратные регистры при каждом переключении центрального процессора с одного процесса на другой. Для этих машин наличие небольших страниц означает увеличение времени, необходимого для загрузки регистров страниц при уменьшении размера страницы. Более того, при уменьшении размера страницы увеличивается пространство, занимаемое таблицей страниц.

Последнее утверждение может быть проанализировано с математической точки зрения. Пусть средний размер процесса будет составлять *s* байт, а размер страницы — *p* байт. Кроме этого предположим, что на каждую страничную запись требуется *e* байт. Тогда приблизительное количество страниц, необходимое каждому процессу, будет равно s/p, что займет *se/p* байт пространства таблицы страниц. Из-за внутренней фрагментации неиспользуемое пространство памяти в последней странице процесса будет равно p/2. Таким образом, общие издержки на таблицу страниц и внутреннюю фрагментацию будут получены за счет суммирования этих двух величин:

Издержки = *se/p* + p/2.

Первое слагаемое (размер таблицы страниц) будет иметь большее значение при небольшом размере страницы. Второе слагаемое (внутренняя фрагментация) будет иметь большее значение при большом размере страницы. Оптимальный вариант находится где-то посередине. Если взять первую производную по переменной *p* и приравнять ее к нулю, то мы получим уравнение

-se/p2 + 1/2 = 0.

Из этого уравнения можно вывести формулу, дающую оптимальный размер страницы (с учетом только потерь памяти на фрагментацию и на размер таблицы страниц). Результат будет следующим:

**р = л/2хё.**

Для 5 = 1 Мбайт и *е* = 8 байт на каждую запись в таблице страниц оптимальный размер страницы будет 4 Кбайт. Имеющиеся в продаже компьютеры используют размер страницы от 512 байт до 64 Кбайт. Раньше чаще всего использовался размер 1 Кбайт, но сейчас чаще всего встречается размер страницы 4 Кбайт.

**3.5.4. Разделение пространства команд и данных**

У многих компьютеров имеется единое адресное пространство, в котором, как показано на рис. 3.22, *а,* содержатся и программы и данные. При довольно большом объеме этого пространства все работает нормально. Но зачастую объем адресного пространства слишком мал, что заставляет программистов как-то выкручиваться, чтобы поместить в него все необходимое.

**Данные**

**Программа**

**Единое**

**адресное пространство**

232

**1-пространство 0-пространство**

232

**Программа -**

**} Неиспользованные
страницы**

**Данные**

Рис. 3.22. Адресное пространство: *а* — единое; *б* — отдельные адресные пространства

команд (I) и данных ф)

Одно из решений, впервые примененное на шестнадцатиразрядной машине РБР-11, заключалось в использовании отдельных адресных пространств для команд (текста программы) и данных, называемых 1-пространством и Б-пространством соответственно (рис. 3.22, *б).* Каждое пространство простирается от 0 до некоторого максимума, обычно 216 - 1 или 232 - 1. Компоновщик должен знать о том, что используются отдельные I- и Б-пространства, поскольку при их использовании данные переносятся на виртуальный адрес 0, а не начинаются сразу после программы.

На компьютерах такой конструкции страничную организацию памяти могут иметь оба пространства независимо друг от друга. У каждого из них имеется собственная таблица страниц с собственным отображением виртуальных страниц на физические страничные блоки. Когда аппаратуре требуется извлечь команду, она знает, что для этого нужно использовать 1-пространство и таблицу страниц этого 1-пространства. Точно так же обращение к данным должно вестись через таблицу страниц Б-пространства. Кроме этих тонкостей, наличие отдельных I- и Б-пространств не приводит к каким- то особым осложнениям для операционной системы и при этом удваивает доступное адресное пространство.

Поскольку теперь адресные пространства стали большими, серьезные проблемы, связанные с их размерами, ушли в прошлое. Но даже сегодня разделение



1. и D-пространств встречается довольно часто. Правда, вместо обычных адресных пространств теперь это разделение используется в кэше L1, который до сих пор испытывает дефицит памяти.
2. **Совместно используемые страницы**

Еще одним вопросом разработки является совместное использование ресурсов. В больших многозадачных системах одновременное использование несколькими пользователями одной и той же программы является обычным делом. Даже отдельный пользователь может запускать несколько программ, использующих одну и ту же библиотеку. При этом вполне очевидна эффективность совместного использования страниц, чтобы избежать одновременного присутствия в памяти двух копий одной и той же страницы. Проблема в том, что не все страницы могут использоваться совместно. В частности, страницы, предназначенные только для чтения, например содержащие текст программы, могут использоваться совместно, а совместное использование страниц с данными связано с дополнительными сложностями.

Если поддерживаются отдельные I- и D-пространства, задача совместного использования программ становится относительно простой за счет наличия одного или нескольких процессов, использующих одну и ту же таблицу страниц для своих I-пространств, но разные таблицы страниц для своих D-пространств. Обычно в реализациях, поддерживающих совместное использование страниц таким образом, таблица страниц является структурой данных, независимой от таблицы процессов. При этом, как показано на рис. 3.23, каждый процесс в своей таблице процесса имеет два указателя: один на таблицу страниц I-пространства, другой на таблицу страниц D-пространства. Когда планировщик процессов выбирает запускаемый процесс, он использует эти указатели для определения местонахождения таблиц страниц и настройки с их помощью диспетчера памяти (MMU). Процессы могут совместно использовать программы (или иногда библиотеки) даже в отсутствие отдельных I- и D-пространств, но для этого используется более сложный механизм.

Когда два и более процесса совместно используют один и тот же код, возникает проблема совместно используемых страниц. Предположим, что два процесса, *A* и B, запускают редактор и совместно используют его страницы. Если планировщик примет решение удалить процесс *A* из памяти, пожертвовав всеми его страницами и заполнив опустевшие страничные блоки какой-нибудь другой программой, это приведет к тому, что процесс *B* сгенерирует большое количество ошибок отсутствия страницы и вернет эти страницы назад.

Также при остановке процесса *A* необходимо иметь возможность определить, какие страницы все еще используются, чтобы их дисковое пространство случайно не оказалось освобожденным. Просмотр всех таблиц страниц для того, чтобы определить совместное использование страницы, — обычно очень затратная операция, поэтому для отслеживания совместно используемых страниц понадобится специальная структура данных, особенно если предметом совместного использования является отдельная страница (или ряд страниц), а не вся таблица страниц.

Совместное использование данных является более сложной задачей, чем совместное использование кода, но и она не является невыполнимой. В частности, в UNIX после системного вызова *fork* родительский и дочерний процессы вынуждены совместно использовать как текст программы, так и данные. В системах со страничной организацией

**Программа Данные-1 Данные-2**

**Таблицы страниц**

Рис. 3.23. Два процесса, совместно использующие одну и ту же программу,
имеют общие таблицы страниц

памяти часто практикуется предоставление каждому из таких процессов собственной таблицы страниц и наличие у них обоих указателя на один и тот же набор страниц. Таким образом, при выполнении системного вызова *fork* копирования страниц не происходит. Тем не менее все страницы с данными отображаются для каждого процесса как страницы только для чтения — READ ONLY.

Пока оба процесса только читают свои данные, не внося в них изменений, эта ситуация может продолжаться. Как только какой-нибудь из процессов обновит слово памяти, нарушение защиты режима только для чтения приведет к системному прерыванию операционной системы. После чего делается копия вызвавшей эту ситуацию страницы, чтобы у каждого процесса имелся собственный экземпляр. Теперь для обеих копий устанавливается режим чтения-записи — READ-WRITE, поэтому следующие записи в каждую копию происходят без системного прерывания. Эта стратегия означает, что те страницы, которые никогда не изменяются (включая все страницы программы), не нуждаются в копировании. Копирование требуется только для тех страниц, которые на самом деле подвергаются изменениям. Этот подход, названный **копированием при записи** (copy on write), повышает производительность за счет сокращения объема копирования.

1. **Совместно используемые библиотеки**

Совместное использование может быть с другой степенью структурированности. Если программа будет запущена дважды, то большинство операционных систем автоматически организуют совместное использование текстовых страниц, чтобы в памяти была только одна копия. Текстовые страницы всегда используются в режиме только для чтения, поэтому проблем не возникает. В зависимости от операционной системы каждый

процесс может получить собственную частную копию страниц с данными или же они могут совместно использоваться и иметь пометку «Только для чтения». Если какой- нибудь процесс изменяет страницу данных, для него будет создана частная копия, то есть будет использована копия, пригодная для записи.

В современных системах имеется множество больших библиотек, используемых многими процессами, к примеру множество библиотек ввода-вывода и графических библиотек. Статическая привязка всех этих библиотек к каждой исполняемой программе на диске сделала бы их еще более раздутыми, чем есть на самом деле.

Вместо этого должна использоваться общая технология **совместно используемых** (**общих**) **библиотек**, которые в Windows называются динамически подключаемыми библиотеками (Dynamic Link Libraries (**DLL**)). Чтобы сделать идею совместно используемой библиотеки более понятной, рассмотрим сначала традиционную компоновку. При компоновке программы в команде компоновщику указывается один или несколько объектных файлов и, возможно, несколько библиотек, как, например, в команде UNIX

ld \*.o -lc -lm

которая компонует все файлы с расширением .o (object), имеющиеся в текущем каталоге, а затем сканирует две библиотеки, /usr/lib/libc.a и /usr/lib/libm.a. Любые функции, вызываемые в объектных файлах, но не присутствующие в них (например, *printf),* называются **неопределенными внешними функциями** и выискиваются в библиотеках. Если они найдены, то их включают в исполняемый двоичный файл. Любые вызываемые, но не присутствующие в них функции также становятся неопределенными внешними функциями. К примеру, функции *printf* требуется функция *write,* поэтому, если функция *write* еще не включена, компоновщик будет ее разыскивать и, как только найдет, включит в двоичный файл. Когда компоновка завершится, исполняемый двоичный файл, записываемый на диск, будет содержать все необходимые функции. Имеющиеся в библиотеке, но невостребованные функции в него не включаются. Когда программа загружается в память и выполняется, в ней содержатся все необходимые ей функции.

Теперь предположим, что обычная программа использует 20-50 Мбайт функций, относящихся к графике и пользовательскому интерфейсу. Статически скомпонованные сотни программ со всеми этими библиотеками будут тратить впустую громадный объем дискового пространства, а также пространства оперативной памяти, как только они будут загружены, поскольку у системы не будет способа узнать о том, что она может использовать эти библиотеки как общие. И тут на сцену выходят совместно используемые библиотеки. Когда программа скомпонована с учетом совместного использования библиотек (что несколько отличается от статической компоновки), вместо включения реально вызываемых функций компоновщик включает небольшую подпрограмму-заглушку, которая в процессе исполнения привязывается к вызываемой функции. В зависимости от системы или от особенностей конфигурации совместно используемые библиотеки загружаются либо при загрузке программы, либо когда присутствующие в них функции вызываются в первый раз. Разумеется, если совместно используемая библиотека уже загружена другой программой, то нет нужды загружать ее повторно — именно в этом и заключается весь смысл. Следует заметить, что при загрузке или использовании общей библиотеки вся библиотека разом в память не считывается. Она загружается постранично, по мере надобности, поэтому функции, которые не были вызваны, в оперативную память не переносятся.

Сделать исполняемые файлы меньшими по объему и сэкономить пространство памяти помогает еще одно преимущество совместно используемых библиотек: если функция, находящаяся в общей библиотеке, обновляется с целью устранения ошибки, то перекомпилировать программу, которая ее вызывает, не нужно. Старые двоичные файлы сохраняют свою работоспособность. Это свойство приобретает особое значение для коммерческого программного обеспечения, код которого не доставляется клиенту. Например, если корпорация Microsoft находит и исправляет ошибку, влияющую на безопасность системы в некой стандартной библиотеке DLL, программа обновления — *Windows Update* — загрузит новую DLL и заменит ею старую библиотеку, и все программы, использующие данную DLL, будут при следующем запуске автоматически использовать новую версию.

Но совместно используемые библиотеки пришли к нам с одной небольшой проблемой, требующей решения. Эта проблема показана на рис. 3.24. Здесь отображены два процесса, совместно использующие библиотеку размером 20 Кбайт (предположим, что каждый ее блок занимает 4 Кбайт). Но библиотека в каждом процессе располагается по разным адресам, по-видимому, потому что сами программы не совпадают по размеру. В процессе 1 библиотека размещается, начиная с адреса 36 К; в процессе 2 ее размещение начинается с адреса 12 К. Предположим, первое, что должна сделать первая функция библиотеки, — перейти в библиотеке к адресу 16. Если библиотека не использовалась совместно, она может быть перемещена на лету в процессе загрузки, поэтому переход (в процессе 1) может быть осуществлен на виртуальный адрес 36 К + 16. Следует заметить, что физический адрес оперативной памяти, по которому размещается библиотека, не имеет значения, пока все страницы отображаются с виртуальных на физические адреса аппаратурой диспетчера памяти — MMU.

Но как только библиотека начинает использоваться совместно, перемещение на лету уже работать не будет. В конце концов, когда первая функция вызывается

**36К**

о

**Процесс 1 Оперативная Процесс 2**

**память**

Рис. 3.24. Общая библиотека, используемая двумя процессами

процессом 2 (по адресу 12 К), команда перехода вынуждена осуществить его на адрес 12 К + 16, а не на адрес 36 К + 16. В этом и заключается небольшая проблема. Одним из путей ее решения является использование копии при записи и создании новых страниц для каждого процесса, использующего общую библиотеку, и перемещение их на лету во время создания. Но эта схема, разумеется, дискредитирует всю цель совместного использования библиотеки.

Лучшее решение заключается в компиляции совместно используемых библиотек со специальным флажком для компилятора, указывающим этому компилятору не создавать никаких команд, использующих абсолютную адресацию. Вместо этого применяются лишь те команды, которые используют относительную адресацию. Например, почти всегда есть команда, предписывающая переход вперед (или назад) на *п* байтов (в качестве альтернативы той команде, которая дает для перехода конкретный адрес). Эта команда работает правильно независимо от размещения совместно используемой библиотеки в виртуальном адресном пространстве. Проблема может быть решена за счет исключения абсолютной адресации. Код, использующий только относительные смещения, называется **позиционно независимым кодом**.

1. **Отображаемые файлы**

На самом деле совместно используемые библиотеки являются частным случаем более общих объектов, называемых **отображаемыми на память файлами**. Идея состоит в том, что процесс может выдать системный вызов для отображения файла на какую-то часть его виртуального адресного пространства. В большинстве реализаций на момент отображения в память еще не введены никакие страницы, но поскольку мы имеем дело со страницами, они требуют постраничной организации с использованием дискового файла в качестве резервного хранилища. Когда процесс выходит из рабочего состояния или явным образом демонтирует отображение файла, все измененные страницы записываются обратно в файл на диске.

Отображаемые файлы предоставляют альтернативную модель для ввода-вывода. Вместо осуществления операций чтения и записи к файлу можно обращаться как к большому символьному массиву, находящемуся в памяти. В некоторых ситуациях программисты находят эту модель более удобной.

Если два или более процесса одновременно отображаются на один и тот же файл, они могут связываться посредством совместно используемой памяти. Запись, произведенная одним процессом в общую память, становится тут же видна, если другой процесс считывает данные из части своего виртуального адресного пространства, отображенного на файл. Таким образом, данный механизм предоставляет канал между двумя процессами, обладающий высокой пропускной способностью, и он довольно часто используется именно в этом качестве (вплоть до отображения рабочего файла). Теперь вы должны понять, что при доступности отображаемых на память файлов совместно используемые библиотеки могут воспользоваться этим механизмом.

1. **Политика очистки страниц**

Замещение страниц лучше всего работает при наличии достаточного количества свободных страничных блоков, которые могут потребоваться при возникновении ошибки отсутствия страницы. Если заполнен и, более того, изменен каждый страничный блок, то перед помещением в него новой страницы сначала должна быть записана на диск старая страница. Для обеспечения поставки свободных страничных блоков системы замещения страниц, как правило, имеют фоновый процесс, называемый **страничным демоном**, который большую часть времени находится в состоянии спячки, но периодически пробуждается для проверки состояния памяти. Если свободно слишком мало страничных блоков, страничный демон начинает подбирать страницы для выгрузки,

используя какой-нибудь алгоритм замещения страниц. Если эти страницы со времени своей загрузки подверглись изменению, они записываются на диск.

В любом случае предыдущее содержание страницы запоминается. Если одна из выгруженных страниц понадобится опять перед тем, как ее страничный блок будет переписан, она может быть восстановлена из резерва свободных страничных блоков. Сохранение материалов страничных блоков улучшает производительность по сравнению с использованием всей памяти с последующей попыткой найти блок в тот момент, когда в нем возникает необходимость. Как минимум, страничный демон обеспечивает чистоту всех свободных блоков, чтобы не приходилось в спешке записывать их на диск, когда в них возникнет потребность.

Один из способов реализации этой политики очистки предусматривает использование часов с двумя стрелками. Передняя стрелка управляется страничным демоном. Когда она указывает на измененную страницу, эта страница сбрасывается на диск и передняя стрелка перемещается вперед. Когда она указывает на чистую страницу, то происходит только перемещение вперед. Задняя стрелка используется для замещения страниц, как в стандартном алгоритме «часы». Только теперь благодаря работе страничного демона повышается вероятность того, что задняя стрелка попадет на чистую страницу.

1. **Интерфейс виртуальной памяти**

До сих пор в нашем повествовании предполагалось, что виртуальная память вполне обозрима процессами и программистами, то есть все, что они видят, — это большое виртуальное адресное пространство на компьютере с малой (или меньшей) по объему физической памятью. Это верно по отношению ко многим системам, но в некоторых системах программистам доступен контроль над отображением памяти и они могут воспользоваться им нетрадиционными способами, чтобы обогатить поведение программы. В этом разделе мы вкратце рассмотрим некоторые из этих возможностей.

Одним из поводов предоставления программистам контроля над отображением памяти является разрешение одному или нескольким процессам совместно использовать одну и ту же память, иногда весьма сложными способами. Если программисты могут присваивать имена областям памяти, то появляется возможность одному процессу предоставить другому процессу имя области памяти, чтобы этот процесс мог также отображаться на нее. Когда два (или несколько) процесса совместно используют одни и те же страницы, появляется возможность использования общего высокоскоростного канала: один процесс ведет запись в общую память, а другой процесс считывает из нее данные.

Совместное использование страниц может быть применено также для реализации высокопроизводительной системы сообщений. Как правило, когда передается сообщение, данные копируются из одного адресного пространства в другое с существенными издержками. Если процессы могут управлять своей таблицей страниц, сообщение может быть передано за счет исключения из таблицы страницы (или страниц), содержащей сообщение, и за счет включения ее в таблицу принимающего процесса. В этом случае должны копироваться только имена, а не все данные.

Еще одна передовая технология управления памятью называется **распределенной памятью совместного доступа**

В ее основе лежит идея, заключающаяся в том, чтобы позволить нескольким процессам через сетевое подключение совместно использовать набор страниц, при этом возможно, но не обязательно, в качестве единого общего линейного диапазона адресов. Когда процесс обращается к странице, которая в данный момент не имеет отображения, у него происходит ошибка отсутствия страницы. Затем обработчик этой ошибки, который может быть в ядре или в пользовательском пространстве, определяет машину, содержащую эту страницу, и посылает ей сообщение с просьбой отключить отображение страницы и переслать ее по сети. По прибытии страницы она отображается, и команда, вызвавшая ошибку, перезапускается.