МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«ВОРОНЕЖСКИЙ ГОСУДАРСТВЕННЫЙ
УНИВЕРСИТЕТ»

Г.Э. Вощинская, М.А. Артемов

ОПЕРАЦИОННЫЕ СИСТЕМЫ

Часть 2

Учебно-методическое пособие для вузов

Издательско-полиграфический центр
Воронежского государственного университета
2012

Утверждено научно-методическим советом факультета прикладной математики, информатики и механики 29 мая 2012 г., протокол № 10

Рецензент доц. кафедры ядерной физики физического ф-та, канд. физ.-мат. наук К.С. Рыбак

Учебно-методическое пособие подготовлено на кафедре программного обеспечения и администрирования информационных систем факультета прикладной математики, информатики и механики Воронежского государственного университета.

Рекомендуется для студентов 3-го курса дневного отделения

Для специальности 010503 – Математическое обеспечение и администрирование информационных систем

Содержание

ТУПИКИ

Пользуясь элементарными средствами связи, например, семафорами, процессы могут для синхронизации своих действий блокировать друг друга и снимать блокировку один с другого. Однако если элементарными синхронизирующими операциями пользоваться неосторожно, то может возникнуть ситуация, называемая тупиком. Существует формальное определение понятия «тупиковая ситуация» в терминах простой модели, описанной Холтом.

Пусть система представляет собой множество состояний и множество процессов, где каждый процесс есть функция, отображающая состояния в состояния.

Процесс заблокирован в некотором состоянии, если он не может работать, когда система находится в этом состоянии.

Процесс находится в тупике в некотором состоянии, если он заблокирован в данном состоянии системы и во всех состояниях, в которые система может перейти в будущем.

Состояние *безопасно*, если никакой процесс не может отобразить его в тупиковое.

В системе со многими различными видами ресурсов тупики – это трудная проблема.

Условия наличия тупика

Коффман, Элфик, Шошани сформулировали четыре **необходимых** условия наличия тупика:

- *Условие взаимоисключения* – процессы требуют предоставления им права монопольного управления ресурсами, которые им выделяются.
- *Условие ожидания ресурсов* – процессы удерживают за собой уже выделенные ресурсы, требуя выделения дополнительных ресурсов.
- *Условие неперераспределяемости* – ресурсы нельзя отобрать у процесса, пока он не завершит работу с ними.
- *Условие кругового ожидания* – существует круговая цепь процессов, в которой каждый удерживает один или более ресурсов, требующихся другому.

Можно выделить четыре направления исследований по проблеме тупиков:

- предотвращение тупиков,
- обход тупиков,
- обнаружение тупиков,
- восстановление после тупиков.

Предотвращение тупиков

При предотвращении тупиков целью является обеспечение условий, исключающих возможность возникновения тупиковых ситуаций. Поэтому,

когда какой-нибудь процесс делает запрос, который может привести к тупику, система принимает меры к тому, чтобы избежать опасного состояния: либо не удовлетворяет этот запрос, либо отбирает ресурс у другого процесса, чтобы избежать возможного попадания в тупик. Достоинство такого подхода в полном исключении тупиков. Недостаток: такой подход часто приводит к нерациональному использованию ресурсов, да и сам предотвращающий алгоритм может внести большие накладные расходы.

Обход тупиков

Цель средств обхода тупиков заключается в том, чтобы можно было предусмотреть менее жесткие ограничения, чем в случае предотвращения тупиков. Методы обхода тупиков учитывают возможность возникновения тупика, однако в этом случае принимаются меры по аккуратному обходу тупика.

Обнаружение тупиков

Методы обнаружения тупиков применяются в системах, допускающих возможность возникновения тупиков. Когда это происходит, система обнаруживает тупик программным путём и принимает меры для вывода из тупика (например, перераспределяя ресурсы). Выход из тупика может выполняться автоматически или под управлением оператора.

Восстановление после тупиков

Методы восстановления после тупиков применяются для устранения тупиковых ситуаций. Например:

- восстановление при помощи принудительной выгрузки;
- восстановление через откат;
- восстановление путем уничтожения одного или нескольких процессов.

Алгоритмы предотвращения тупиков

Рассмотрим некоторые алгоритмы предотвращения тупиков.

Выделение всех необходимых ресурсов

Процесс получает все необходимые ему ресурсы перед началом работы. В этом случае возникновение тупика исключается, но нет разделения ресурсов и использование ресурсов нерационально.

Выделение ресурсов в порядке присвоенных номеров

Все разделяемые ресурсы в системе пронумерованы. Процесс запрашивает ресурсы по возрастанию номеров. Он не может запрашивать следующий ресурс, если предыдущий запрос не удовлетворен. Например, процесс использует сначала ресурс 3, затем ресурс 5, затем ресурс 2. Тогда, чтобы получить ресурс 3, он должен сначала запросить и получить ресурс 2, а затем запрашивать ресурс 3. Ресурс 5 он может получить позже, когда он

ему понадобится. Этот алгоритм допускает разделение ресурсов, условия выделения ресурсов менее жесткие, чем в предыдущем способе, но, в некоторых случаях, когда порядок работы процесса с ресурсами не совпадает с нумерацией ресурсов, процесс должен захватывать и удерживать ресурс заранее, хотя использовать его он будет позже.

Метод Габермана

Системе требуются

a) ориентированный граф, в котором узлы соответствуют процессам, а дуга проводится от узла i к узлу j, если процесс j может запросить ресурс, который запрашивает процесс i;

b) предварительная информация о ресурсах, необходимых каждому процессу. Она хранится в таблице, в которой строки соответствуют процессам, а столбцы – ресурсам;

c) таблица учета выделенных ресурсов.

Правило Габермана гласит: состояние является опасным, если граф содержит цикл.

Алгоритм вызывается при каждом запросе и возврате ресурса.

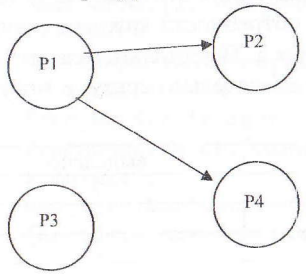
Например.

Пусть в системе выполняются процессы P1, P2, P3, P4, и ими используются ресурсы A, B,C, D. В таблице для каждого процесса 1 отмечены ресурсы, которые ему могут понадобиться.

	A	B	C	D
P1	1	1		
P2	1	1	1	
P3		1	1	1
P4	1		1	1

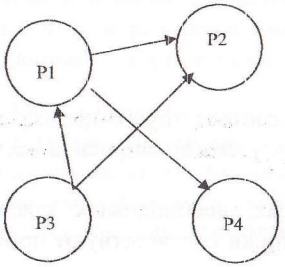
Предположим следующий сценарий запросов.

Пусть P1 запрашивает A. Ресурс свободен. Ресурс A могут еще запросить процессы P2 и P4. В графе проводятся дуги от P1 к P2 и от P1 к P4. Цикла нет – ресурс выделяется.



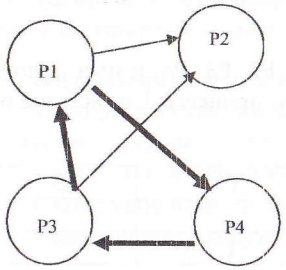
Пусть P2 запрашивает A. Ресурс занят – процесс P2 блокируется.

Пусть P3 запрашивает B. Ресурс свободен. Но ресурс B могут запросить также процессы P1 и P2. В графе проводятся дуги от P3 к P1 и от P3 к P2.



Цикла нет – ресурс выделяется.

Пусть P4 запрашивает D. Ресурс свободен. Но ресурс D может запросить процесс P3. В графе проводится дуга от P4 к P3.



Есть цикл – ресурс не выделяется. Процесс P4 блокируется. Граф возвращается в предыдущее состояние.

Алгоритм банкира

Этот алгоритм используется для распределения делимых ресурсов. Рассмотрим его идею на примере. Пусть в системе имеется 10 единиц некоторого ресурса и работают 3 процесса, использующих этот ресурс. В таблице содержится информация о максимальной потребности каждого процесса и о количестве уже выделенных единиц ресурса. Предполагается, что процессы используют необходимые ресурсы не обязательно сразу, а могут запрашивать их по частям.

процесс	max потр.	выделено
P1	4	2
P2	7	3
P3	8	2

Рассмотрим два варианта запросов из данного состояния.

Пусть P1 запрашивает 2 устройства.

процесс	max потр.	выделено
P1	4	4
P2	7	3
P3	8	2

P1 получит все нужные ему ресурсы, завершит работу и освободит полученные им ресурсы, тогда сможет завершиться P2, а после его завершения и P3. Т.е. этот запрос безопасный.

Пусть P2 запрашивает 2 устройства.

процесс	max потр.	выделено
P1	4	2
P2	7	5
P3	8	2

Оставшегося одного устройства недостаточно для завершения ни одного из процессов. Система оказалась в тупике. Значит система, не допускающая опасные состояния, должна отклонить такой запрос.

Ниже приводится алгоритм банкира, предложенный Дейкстрой.

```
program alg_banker;
const
  num_device=100;//общее число устройств
  num_proc=50;//общее число процессов
var
  max_want:array[1..num_proc] of integer;
  //максимальная потребность процессов
  allot:array[1..num_proc] of integer;
  //выделено устройств
  rest:array[1..num_proc] of integer;
  //остаток
  not_completed:array[1..num_proc] of boolean;
  //=true, если процесс может не завершиться
  free_device:integer;
  //количество свободных устройств
  i:integer;
  was_free:boolean;//признак освобождения устройств
  {алгоритм применяется после предварительного
```

```
выделения устройств}
begin
  free_device:=num_device;
for i:=1 to num_proc do
    begin
      free_device:=free_device-allot[i];
      not_completed[i]:=true;
      rest[i]:=max_want[i]-allot[i];
    end;
repeat
   was_free:=false;
    for i:=1 to num_proc do
        if not_completed[i] and (rest[i]<=free_device) do
            begin
                not_completed[i]:=false;
                 free_device:=free_device+allot[i];
                was_free:=true;
            end;
until not was_free;
if free_device=num_device then State_is_safety else
State_is_not_safety;
end.
```

Тупики как критический фактор для будущих систем

В современных больших системах тупики являются критическим фактором, а в будущих системах роль этого фактора еще более возрастет, т.к.

- процессы будут в гораздо большей степени ориентированы на асинхронную параллельную работу. Мультипроцессорные архитектуры и параллельные вычисления займут доминирующее положение;
- в этих системах будет реализовано преимущественно динамическое распределение ресурсов;
- среди разработчиков ОС растет тенденция рассматривать данные как ресурс.

Управление памятью

Память является важнейшим ресурсом, требующим тщательного управления со стороны мультипрограммной операционной системы. Распределению подлежит вся оперативная память, не занятая операционной системой. Обычно ОС располагается в самых младших адресах, однако может занимать и самые старшие адреса. Функциями ОС по управлению памятью являются:

- отслеживание свободной и занятой памяти;

- выделение памяти процессам;
- освобождение памяти при завершении процессов;
- вытеснение процессов из оперативной памяти на диск, когда размеры основной памяти недостаточны для размещения в ней всех процессов, и возвращение их в оперативную память, когда в ней освобождается место;
- настройка адресов программы на конкретную область физической памяти.

Типы адресов

Для идентификации переменных и команд используются символьные имена (метки), виртуальные адреса и физические адреса (рис. 1).

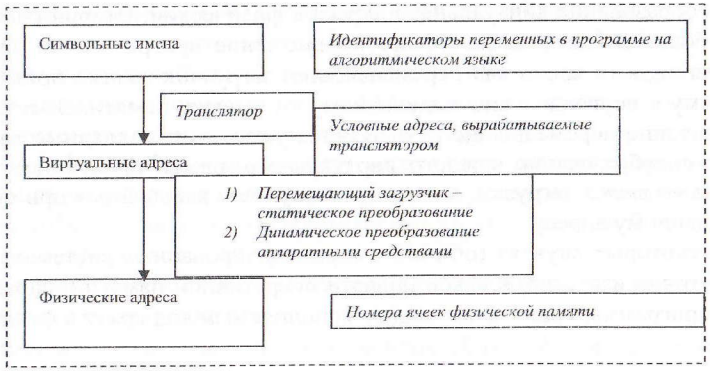


Рис. 1. Типы адресов

Символьные имена присваивает пользователь при написании программы на алгоритмическом языке или ассемблере.

Виртуальные адреса вырабатывает транслятор, переводящий программу на машинный язык. Так как во время трансляции в общем случае неизвестно, в какое место оперативной памяти будет загружена программа, то транслятор присваивает переменным и командам виртуальные (условные) адреса, обычно считая по умолчанию, что программа будет размещена, начиная с нулевого адреса. Совокупность виртуальных адресов процесса называется виртуальным адресным пространством. Каждый процесс имеет собственное виртуальное адресное пространство. Максимальный размер виртуального адресного пространства ограничивается разрядностью адреса, присущей данной архитектуре компьютера, и, как правило, не совпадает с объемом физической памяти, имеющимся в компьютере.

Физические адреса соответствуют номерам ячеек оперативной памяти, где в действительности расположены или будут расположены переменные и команды. Переход от виртуальных адресов к физическим может осуществ-

ляться двумя способами. В первом случае замену виртуальных адресов на физические делает специальная системная программа – перемещающий загрузчик. Перемещающий загрузчик на основании имеющихся у него исходных данных о начальном адресе физической памяти, в которую предстоит загружать программу, и информации, предоставленной транслятором об адресно-зависимых константах программы, выполняет загрузку программы, совмещая ее с заменой виртуальных адресов физическими.

Второй способ заключается в том, что программа загружается в память в неизмененном виде в виртуальных адресах, при этом операционная система фиксирует смещение действительного расположения программного кода относительно виртуального адресного пространства. Во время выполнения программы при каждом обращении к оперативной памяти выполняется преобразование виртуального адреса в физический. Второй способ является более гибким, он допускает перемещение программы во время ее выполнения, в то время как перемещающий загрузчик жестко привязывает программу к первоначально выделенному ей участку памяти. Вместе с тем использование перемещающего загрузчика уменьшает накладные расходы, так как преобразование каждого виртуального адреса происходит только один раз во время загрузки, а во втором случае – каждый раз при обращении по данному адресу.

В некоторых случаях (обычно в специализированных системах), когда заранее точно известно, в какой области оперативной памяти будет выполняться программа, транслятор выдает исполняемый код сразу в физических адресах.

Методы распределения памяти без использования дискового пространства

Все методы управления памятью могут быть разделены на два класса: методы, которые используют перемещение процессов между оперативной памятью и диском, и методы, которые не делают этого.

Распределение памяти фиксированными разделами

Самым простым способом управления оперативной памятью является разделение ее на несколько разделов фиксированной величины. Это может быть выполнено вручную оператором во время старта системы или во время ее генерации. Очередная задача, поступившая на выполнение, помещается либо в общую очередь, либо в очередь к некоторому разделу.

Подсистема управления памятью в этом случае выполняет следующие задачи:

- сравнивая размер программы, поступившей на выполнение, и свободных разделов, выбирает подходящий раздел;
- осуществляет загрузку программы и настройку адресов.

При очевидном преимуществе – простоте реализации – данный метод имеет существенный недостаток – жесткость. Так как в каждом разделе может выполняться только одна программа, то уровень мультипрограммирования заранее ограничен числом разделов независимо от того, какой размер имеют программы. Даже если программа имеет небольшой объем, она будет занимать весь раздел, что приводит к неэффективному использованию памяти. С другой стороны, даже если объем оперативной памяти машины позволяет выполнить некоторую большую программу, разбиение памяти на разделы может не позволить сделать этого.

Распределение памяти разделами переменной величины

В этом случае память машины не делится заранее на разделы. Сначала вся память свободна. Каждой вновь поступающей задаче выделяется необходимая ей память. Если достаточный объем памяти отсутствует, то задача не принимается на выполнение и стоит в очереди. После завершения задачи память освобождается, и на это место может быть загружена другая задача. Таким образом, в произвольный момент времени оперативная память представляет собой случайную последовательность занятых и свободных участков (разделов) произвольного размера.

Задачами операционной системы при реализации данного метода управления памятью является:

- ведение таблиц свободных и занятых областей, в которых указываются начальные адреса и размеры участков памяти;
- при поступлении новой задачи – анализ запроса, просмотр таблицы свободных областей и выбор раздела, размер которого достаточен для размещения поступившей задачи;
- загрузка задачи в выделенный ей раздел и корректировка таблиц свободных и занятых областей;
- после завершения задачи корректировка таблиц свободных и занятых областей.

Программный код не перемещается во время выполнения, то есть может быть проведена единовременная настройка адресов посредством использования перемещающего загрузчика.

Выбор раздела для вновь поступившей задачи может осуществляться по разным правилам, таким, например, как «первый попавшийся раздел достаточного размера», или «раздел, имеющий наименьший достаточный размер», или «раздел, имеющий наибольший достаточный размер». Все эти правила имеют свои преимущества и недостатки.

По сравнению с методом распределения памяти фиксированными раз-

делами данный метод обладает гораздо большей гибкостью, но ему присущ очень серьезный недостаток – фрагментация памяти. *Фрагментация* – это наличие большого числа несмежных участков свободной памяти очень маленького размера (фрагментов). Настолько маленького, что ни одна из вновь поступающих программ не может поместиться ни в одном из участков, хотя суммарный объем фрагментов может составить значительную величину, намного превышающую требуемый объем памяти.

Перемещаемые разделы

Одним из методов борьбы с фрагментацией является перемещение всех занятых участков в сторону старших либо в сторону младших адресов, так, чтобы вся свободная память образовывала единую свободную область. В дополнение к функциям, которые выполняет ОС при распределении памяти переменными разделами, в данном случае она должна еще время от времени копировать содержимое разделов из одного места памяти в другое, корректируя таблицы свободных и занятых областей. Эта процедура называется «сжатием». Сжатие может выполняться либо при каждом завершении задачи, либо только тогда, когда для вновь поступившей задачи нет свободного раздела достаточного размера. В первом случае требуется меньше вычислительной работы при корректировке таблиц, а во втором – реже выполняется процедура сжатия. Так как программы перемещаются по оперативной памяти в ходе своего выполнения, то преобразование адресов из виртуальной формы в физическую, должно выполняться динамическим способом.

Хотя процедура сжатия и приводит к более эффективному использованию памяти, она может потребовать значительного времени, что часто перевешивает преимущества данного метода.

Методы распределения памяти с использованием дискового пространства

Понятие виртуальной памяти

Уже достаточно давно пользователи столкнулись с проблемой размещения в памяти программ, размер которых превышал имеющуюся в наличии свободную память. Решением было разбиение программы на части, называемые оверлеями. Нулевой оверлей начинал выполняться первым. Когда он заканчивал свое выполнение, он вызывал другой оверлей. Все оверлеи хранились на диске и перемещались между памятью и диском средствами операционной системы. Однако разбиение программы на части и планирование их загрузки в оперативную память должен был осуществлять программист.

Развитие методов организации вычислительного процесса в этом направлении привело к появлению метода, известного под названием виртуальная память. ***Виртуальным*** называется ресурс, который пользователю или пользовательской программе представляется обладающим свойствами, которыми он в действительности не обладает. Так, например, пользователю может быть предоставлена виртуальная оперативная память, размер которой превосходит всю имеющуюся в системе реальную оперативную память. Пользователь пишет программы так, как будто в его распоряжении имеется однородная оперативная память большого объема, но в действительности все данные, используемые программой, хранятся на одном или нескольких разнородных запоминающих устройствах, обычно на дисках, и при необходимости частями отображаются в реальную память.

Таким образом, ***виртуальная память*** – это совокупность программно-аппаратных средств, позволяющих пользователям писать программы, размер которых превосходит имеющуюся оперативную память; для этого виртуальная память решает следующие задачи:

- размещает данные в запоминающих устройствах разного типа, например, часть программы в оперативной памяти, а часть на диске;
- перемещает по мере необходимости данные между запоминающими устройствами разного типа, например, подгружает нужную часть программы с диска в оперативную память;
- преобразует виртуальные адреса в физические.

Все эти действия выполняются *автоматически,* без участия программиста, то есть механизм виртуальной памяти является прозрачным по отношению к пользователю.

Наиболее распространенными реализациями виртуальной памяти является страничное, сегментное и странично-сегментное распределение памяти, а также свопинг.

Страничное распределение

Виртуальное адресное пространство каждого процесса делится на части одинакового, фиксированного для данной системы размера, называемые *виртуальными страницами*. В общем случае размер виртуального адресного пространства не является кратным размеру страницы, поэтому последняя страница каждого процесса дополняется фиктивной областью.

Вся оперативная память машины также делится на части такого же размера, называемые *физическими страницами (или блоками).*

Размер страницы обычно выбирается равным степени двойки: 512, 1024 и т.д., это позволяет упростить механизм преобразования адресов.

При загрузке процесса часть его виртуальных страниц помещается в оперативную память, а остальные – на диск. Смежные виртуальные страницы не обязательно располагаются в смежных физических страницах. При

загрузке операционная система создает для каждого процесса информационную структуру – таблицу страниц, в которой устанавливается соответствие между номерами виртуальных и физических страниц для страниц, загруженных в оперативную память, или делается отметка о том, что виртуальная страница выгружена на диск. Кроме того, в таблице страниц содержится управляющая информация, такая как признак модификации страницы, признак невыгружаемости (выгрузка некоторых страниц может быть запрещена), признак обращения к странице (используется для подсчета числа обращений за определенный период времени) и другие данные, формируемые и используемые механизмом виртуальной памяти.

При активизации очередного процесса в специальный регистр процессора загружается адрес таблицы страниц данного процесса.

При каждом обращении к памяти происходит чтение из таблицы страниц информации о виртуальной странице, к которой произошло обращение. Если данная виртуальная страница находится в оперативной памяти, то выполняется преобразование виртуального адреса в физический. Если же нужная виртуальная страница в данный момент выгружена на диск, то происходит так называемое *страничное прерывание*. Выполняющийся процесс переводится в состояние ожидания, и активизируется другой процесс из очереди готовых. Параллельно программа обработки страничного прерывания находит на диске требуемую виртуальную страницу и пытается загрузить ее в оперативную память. Если в памяти имеется свободная физическая страница, то загрузка выполняется немедленно, если же свободных страниц нет, то решается вопрос, какую страницу следует выгрузить из оперативной памяти.

В данной ситуации может быть использовано много разных критериев выбора, наиболее популярные из них следующие:

- дольше всего не использовавшаяся страница,
- первая попавшаяся страница,
- страница, к которой в последнее время было меньше всего обращений.

В некоторых системах используется понятие рабочего множества страниц. Рабочее множество определяется для каждого процесса и представляет собой перечень наиболее часто используемых страниц, которые должны постоянно находиться в оперативной памяти и поэтому не подлежат выгрузке.

После того, как выбрана страница, которая должна покинуть оперативную память, анализируется ее признак модификации (из таблицы страниц). Если выталкиваемая страница с момента загрузки была модифицирована, то ее новая версия должна быть переписана на диск. Если нет, то она может быть просто уничтожена, то есть соответствующая физическая страница объявляется свободной.

Рассмотрим механизм преобразования виртуального адреса в физический при страничной организации памяти.

Виртуальный адрес при страничном распределении может быть представлен в виде пары (p, s), где p – номер виртуальной страницы процесса (нумерация страниц начинается с 0), а s – смещение в пределах виртуальной страницы. Учитывая, что размер страницы равен 2 в степени k, смещение s может быть получено простым отделением k младших разрядов в двоичной записи виртуального адреса. Оставшиеся старшие разряды представляют собой двоичную запись номера страницы p.

При каждом обращении к оперативной памяти аппаратными средствами выполняются следующие действия:

1) на основании начального адреса таблицы страниц (содержимое регистра адреса таблицы страниц), номера виртуальной страницы (старшие разряды виртуального адреса) и длины записи в таблице страниц (системная константа) определяется адрес нужной записи в таблице;

2) из этой записи извлекается номер физической страницы;

3) к номеру физической страницы присоединяется смещение (младшие разряды виртуального адреса).

Использование в пункте (3) того факта, что размер страницы равен степени 2, позволяет применить операцию конкатенации (присоединения) вместо более длительной операции сложения, что уменьшает время получения физического адреса, а значит, повышает производительность компьютера.

На производительность системы со страничной организацией памяти влияют временные затраты, связанные с обработкой страничных прерываний и преобразованием виртуального адреса в физический. При часто возникающих страничных прерываниях система может тратить большую часть времени впустую, на свопинг страниц. Чтобы уменьшить частоту страничных прерываний, следовало бы увеличивать размер страницы. Кроме того, увеличение размера страницы уменьшает размер таблицы страниц, а значит, уменьшает затраты памяти. С другой стороны, если страница велика, значит велика и фиктивная область в последней виртуальной странице каждой программы. В среднем на каждой программе теряется половина объема страницы, что в сумме при большой странице может составить существенную величину. Время преобразования виртуального адреса в физический в значительной степени определяется временем доступа к таблице страниц. В связи с этим таблицу страниц стремятся размещать в «быстрых» запоминающих устройствах. Это может быть, например, набор специальных регистров или память, использующая для уменьшения времени доступа ассоциативный поиск и кэширование данных.

Страничное распределение памяти может быть реализовано в упрощенном варианте, без выгрузки страниц на диск. В этом случае все виртуальные страницы всех процессов постоянно находятся в оперативной памяти. Такой вариант страничной организации хотя и не предоставляет пользо-

вателю виртуальной памяти, но почти исключает фрагментацию за счет того, что программа может загружаться в несмежные области, а также того, что при загрузке виртуальных страниц никогда не образуется остатков.

Сегментное распределение

При страничной организации виртуальное адресное пространство процесса делится механически на равные части. Это не позволяет дифференцировать способы доступа к разным частям программы (сегментам), а это свойство часто бывает очень полезным. Например, можно запретить обращаться с операциями записи и чтения в кодовый сегмент программы, а для сегмента данных разрешить только чтение. Кроме того, разбиение программы на «осмысленные» части делает принципиально возможным разделение одного сегмента несколькими процессами. Например, если два процесса используют одну и ту же математическую подпрограмму, то в оперативную память может быть загружена только одна копия этой подпрограммы.

Рассмотрим, каким образом сегментное распределение памяти реализует эти возможности. Виртуальное адресное пространство процесса делится на сегменты, размер которых определяется программистом с учетом смыслового значения содержащейся в них информации. Отдельный сегмент может представлять собой подпрограмму, массив данных и т.п. Иногда сегментация программы выполняется по умолчанию компилятором.

При загрузке процесса часть сегментов помещается в оперативную память (при этом для каждого из этих сегментов операционная система подыскивает подходящий участок свободной памяти), а часть сегментов размещается в дисковой памяти. Сегменты одной программы могут занимать в оперативной памяти несмежные участки. Во время загрузки система создает таблицу сегментов процесса (аналогичную таблице страниц), в которой для каждого сегмента указывается начальный физический адрес сегмента в оперативной памяти, размер сегмента, правила доступа, признак модификации, признак обращения к данному сегменту за последний интервал времени и некоторая другая информация. Если виртуальные адресные пространства нескольких процессов включают один и тот же сегмент, то в таблицах сегментов этих процессов делаются ссылки на один и тот же участок оперативной памяти, в который данный сегмент загружается в единственном экземпляре.

Система с сегментной организацией функционирует аналогично системе со страничной организацией: время от времени происходят прерывания, связанные с отсутствием нужных сегментов в памяти, при необходимости освобождения памяти некоторые сегменты выгружаются, при каждом обращении к оперативной памяти выполняется преобразование виртуального адреса в физический. Кроме того, при обращении к памяти проверяется, разрешен ли доступ требуемого типа к данному сегменту.

Виртуальный адрес при сегментной организации памяти может быть представлен парой (g, s), где g – номер сегмента, а s – смещение в сегменте. Физический адрес получается путем сложения начального физического адреса сегмента, найденного в таблице сегментов по номеру g, и смещения s.

Недостатком данного метода распределения памяти является фрагментация на уровне сегментов и более медленное по сравнению со страничной организацией преобразование адреса.

Странично-сегментное распределение

Как видно из названия, данный метод представляет собой комбинацию страничного и сегментного распределения памяти и, вследствие этого, сочетает в себе достоинства обоих подходов. Виртуальное пространство процесса делится на сегменты, а каждый сегмент в свою очередь делится на виртуальные страницы, которые нумеруются в пределах сегмента. Оперативная память делится на физические страницы. Загрузка процесса выполняется операционной системой постранично, при этом часть страниц размещается в оперативной памяти, а часть на диске. Для каждого сегмента создается своя таблица страниц, структура которой полностью совпадает со структурой таблицы страниц, используемой при страничном распределении. Для каждого процесса создается таблица сегментов, в которой указываются адреса таблиц страниц для всех сегментов данного процесса. Адрес таблицы сегментов загружается в специальный регистр процессора, когда активизируется соответствующий процесс.

Свопинг

Разновидностью виртуальной памяти является свопинг. Загрузка процессора зависит от числа одновременно выполняемых процессов и доли времени, проводимого этими процессами в состоянии ожидания ввода-вывода.

Для загрузки процессора на 90 % достаточно всего трех счетных задач. Однако для того, чтобы обеспечить такую же загрузку интерактивными задачами, выполняющими интенсивный ввод-вывод, потребуются десятки таких задач. Необходимым условием для выполнения задачи является загрузка ее в оперативную память, объем которой ограничен. В этих условиях был предложен метод организации вычислительного процесса, называемый свопингом. В соответствии с этим методом некоторые процессы (обычно находящиеся в состоянии ожидания) временно выгружаются на диск. Планировщик операционной системы не исключает их из своего рассмотрения, и при наступлении условий активизации некоторого процесса, находящегося в области свопинга на диске, этот процесс перемещается в оперативную память. Если свободного места в оперативной памяти не хватает, то выгружается другой процесс.

При свопинге, в отличие от рассмотренных ранее методов реализации виртуальной памяти, процесс перемещается между памятью и диском целиком, то есть в течение некоторого времени процесс может полностью отсутствовать в оперативной памяти. Существуют различные алгоритмы выбора процессов на загрузку и выгрузку, а также различные способы выделения оперативной и дисковой памяти загружаемому процессу.

Иерархия запоминающих устройств. Принцип кэширования данных

Память вычислительной машины представляет собой иерархию запоминающих устройств (внутренние регистры процессора, различные типы сверхоперативной и оперативной памяти, диски, ленты), отличающихся средним временем доступа и стоимостью хранения данных в расчете на один бит (рис. 2). Пользователю хотелось бы иметь и недорогую и быструю память. Кэш-память представляет некоторое компромиссное решение этой проблемы.

Рис. 2. Иерархия ЗУ

Кэш-память – это способ организации совместного функционирования двух типов запоминающих устройств, отличающихся временем доступа и стоимостью хранения данных, который позволяет уменьшить среднее время доступа к данным за счет динамического копирования в «быстрое» ЗУ наиболее часто используемой информации из «медленного» ЗУ.

Кэш-памятью часто называют не только способ организации работы двух типов запоминающих устройств, но и одно из устройств – «быстрое» ЗУ. Оно стоит дороже и, как правило, имеет сравнительно небольшой объем. Важно, что механизм кэш-памяти является прозрачным для пользователя, который не должен сообщать никакой информации об интенсивности использования данных и не должен никак участвовать в перемещении данных из ЗУ одного типа в ЗУ другого типа, все это делается автоматически системными средствами.

Рассмотрим частный случай использования кэш-памяти для уменьшения среднего времени доступа к данным, хранящимся в оперативной памяти. Для этого между процессором и оперативной памятью помещается быстрое ЗУ, называемое просто кэш-памятью. В качестве такового может быть

использована, например, ассоциативная память. Содержимое кэш-памяти представляет собой совокупность записей обо всех загруженных в нее элементах данных. Каждая запись об элементе данных включает в себя адрес, который этот элемент данных имеет в оперативной памяти, и управляющую информацию: признак модификации и признак обращения к данным за некоторый последний период времени.

В системах, оснащенных кэш-памятью, каждый запрос к оперативной памяти выполняется в соответствии со следующим алгоритмом:

1. Просматривается содержимое кэш-памяти с целью определения, не находятся ли нужные данные в кэш-памяти; кэш-память не является адресуемой, поэтому поиск нужных данных осуществляется по содержимому – значению поля «адрес в оперативной памяти», взятому из запроса.

2. Если данные обнаруживаются в кэш-памяти, то они считываются из нее, и результат передается в процессор.

3. Если нужных данных нет, то они вместе со своим адресом копируются из оперативной памяти в кэш-память, и результат выполнения запроса передается в процессор. При копировании данных может оказаться, что в кэш-памяти нет свободного места, тогда выбираются данные, к которым в последний период было меньше всего обращений, для вытеснения из кэш-памяти. Если вытесняемые данные были модифицированы за время нахождения в кэш-памяти, то они переписываются в оперативную память. Если же эти данные не были модифицированы, то их место в кэш-памяти объявляется свободным.

На практике в кэш-память считывается не один элемент данных, к которому произошло обращение, а целый блок данных, это увеличивает вероятность так называемого «попадания в кэш», то есть нахождения нужных данных в кэш-памяти.

Покажем, как среднее время доступа к данным зависит от вероятности попадания в кэш. Пусть имеется основное запоминающее устройство со средним временем доступа к данным t1 и кэш-память, имеющая время доступа t2, очевидно, что $t2 < t1$. Обозначим через t среднее время доступа к данным в системе с кэш-памятью, а через p – вероятность попадания в кэш. По формуле полной вероятности имеем: $t = t1((1 - p) + t2(p)$.

Из нее видно, что среднее время доступа к данным в системе с кэш-памятью линейно зависит от вероятности попадания в кэш и изменяется от среднего времени доступа в основное ЗУ (при $p = 0$) до среднего времени доступа непосредственно в кэш-память (при $p = 1$).

В реальных системах вероятность попадания в кэш составляет примерно 0,9. Высокое значение вероятности нахождения данных в кэш-памяти связано с наличием у данных объективных свойств: пространственной и временной локальности.

• *Пространственная локальность*. Если произошло обращение по некоторому адресу, то с высокой степенью вероятности в ближайшее время произойдет обращение к соседним адресам.

• *Временная локальность*. Если произошло обращение по некоторому адресу, то следующее обращение по этому же адресу с большой вероятностью произойдет в ближайшее время.

Все предыдущие рассуждения справедливы и для других пар запоминающих устройств, например, для оперативной памяти и внешней памяти. В этом случае уменьшается среднее время доступа к данным, расположенным на диске, и роль кэш-памяти выполняет буфер в оперативной памяти.

Поиск файлов

Для поиска заданного каталога, системных каталогов, каталогов, заданных с помощью переменной окружения PATH, или списка каталогов, разделенных точкой с запятой, можно использовать функцию Win32 API SearchPath(). К сожалению, эта функция не выполняет поиск файла среди подкаталогов данного каталога. Для поиска файлов в некотором каталоге и его подкаталогах можно использовать пару функций

```
function FindFirst(const Path: string; Attr: Integer;
var F: TSearchRec): Integer;
```

и

```
function FindNext(var F: TSearchRec): Integer;
```

Запись TSearchRec

Запись TSearchRec определяет данные, необходимые для передачи функциям FindFirst() и FindNext (). Object Pascal определяет эту запись следующим образом:

```
TSearchRec = record
    Time: Integer;
    Size: Integer;
    Attr: Integer;
    Name: TFileName;
    ExcludeAttr: Integer;
    FindHandle: THandle;
    FindData: TWin32FindData;
end;
```

- Поле *Time* содержит время создания или модификации файла,
- поле *Size* — размер файла в байтах,
- поле *Name* хранится имя файла,

- поле *Attr* содержит один или несколько атрибутов файла, перечисленных ниже.

Атрибуты файла

Атрибут	Значение	Описание
faReadOnly	$01	Файл, предназначенный только для чтения
faHidden	$02	Скрытый файл
faSysFile	$04	Системный файл
faVolumelD	$08	Метка тома
faDirectory	$10	Каталог
faArchive	$20	Архивный файл
faAnyFile	$3F	Любой файл

Поля FindHandle и ExcludeAttr используются функциями FindFirst() и FindNext() для внутренних нужд, поэтому нет необходимости вникать в их назначение.

Функции FindFirst() и FindNext() принимают путь в качестве параметра, который содержит символы шаблона (например, выражение C:\DELPHI 5\BIN* .EXE означает все файлы с расширением EXE в каталоге C:\DELPHI 5\BIN\). Параметр Attr определяет атрибуты файла, по которым следует проводить поиск. Если, например, вы хотите найти только системные файлы, следует вызывать функции FindFirst() и/или FindNext() следующим образом:

```
FindFirst(Path, faSysFile, SearchRec);
```

Запись TWin32FindData содержит дополнительную информацию о найденном файле или подкаталоге и определяется следующим образом:

```
TWin32FindData=record
    dwFileAttributes:DWORD;
    ftCreationTime: TFileTime;
    ftLastAccessTime: TFileTime;
    ftLastWriteTime: TFileTime;
    nFileSizeHigh: DWORD;
    nFileSizeLow: DWORD;
    dwReserved0: DWORD;
    dwReserved1: DWORD;
    cFileName: array [0..MAX_PATH-1] of AnsiChar;
    cAlternateFileName: array [0..13] of AnsiChar;
end;
```

Значения полей записи TWin32FindData

Поле	Значение
dwFileAttributes	Атрибуты найденного файла
ftCreationTime	Время создания файла
ftLastAccessTime	Время последнего доступа к файлу
ftLastWriteTime	Время последней модификации файла
nFileSizeHigh	Старшие разряды (старшее двойное слово DWORD) размера файла в байтах. Если размер файла не превышает MAXWORD, то это значение равно 0
nFileSizeLow	Младшие разряды(младшее двойное слово DWORD) размера файла в байтах
dwReserved0	В данный момент не используется – зарезервировано
dwReserved1	В данный момент не используется – зарезервировано
cFileName	Имя файла в виде строки с ограничивающим нуль – символом
cAlternateFileName	Имя файла в формате 8.3, усечение длинного имени

Для проверки атрибутов используется функция

```
DWORD GetFileAttributes (LPCTSTR );
```

Для установки атрибутов можно использовать функцию

```
BOOL SetFileAttributes (LPCTSTR, DWORD);
```

Ниже приведен листинг приложения, реализованного в среде DELPHI, для поиска файлов по заданной маске в заданном каталоге, включая все вложенные подкаталоги.

```
unit MainFrm;
interface
uses
Windows, Messages, SysUtils, Classes, Graphics, Controls,
Forms, Dialogs, StdCtrls, FileCtrl, Grids, Outline,
DirOutln;
type TMainForm = class (TForm)
dcbDrives: TDriveComboBox;
edtFileMask: TEdit;
lblFileMask: TLabel;
btnSearchForFiles: TButton;
lbFiles: TListBox;
dolDirectories: TDirectoryOutline;
procedure btnSearchForFilesClickfSender: TObject);
procedure dcbDrivesChange(Sender: TObject); private
FFileName: String;
function GetDirectoryName(Dir: String): String;
```



```
procedure FindFiles(APath: String);
end;
var MainForm: TMainForm;
implementation {$R *.DFM}
function TMainForm.GetDirectoryName(Dir: String): String;
{Эта функция форматирует имя каталога таким образом, чтобы
оно в качестве последнего символа содержало обратную косую
черту (\). }
begin
if Dir[Length(Dir)]<> '\ ' then
Result := Dir+'\ '
else
Result := Dir;
end;
procedure TMainForm.FindFiles(APath: String);
{ эта процедура вызывается рекурсивно для выполнения поиска
заданного маской файла в текущем каталоге и его подкаталогах.}
var
FSearchRec,
DSearchRec: TSearchRec;
FindResult: integer;

function IsDirNotation(ADirName: String): Boolean;
begin
Result := (ADirName = '.') or (ADirName = '..');
   end;
begin
APath := GetDirectoryName(APath);  // Получаем имя каталога
{ Находим первое вхождение заданного имени файла. }
FindResult := FindFirst(APath+FFileName,faAnyFile+faHidden+
faSysFile+faReadOnly,FSearchRec);
try
{ Продолжаем искать файлы в соответствии с заданной маской.
При обнаружении искомого файла добавляем в список его имя и
путь.}
while FindResult = 0 do
begin
IbFiles.Items.Add(LowerCase(APath+FSearchRec.Name));
FindResult := FindNext(FSearchRec);
end;
{ Теперь просмотрим подкаталоги текущего каталога. Для этого
воспользуемся функцией FindFirst для циклического просмотра
каждого каталога, а затем снова вызовем функцию FindFiles
(т.е. себя же). Этот рекурсивный процесс будет продолжаться
```

```
до тех пор, пока не будут просмотрены все подкаталоги. }
FindResult := FindFirst(APath+'*.*', faDirectory, DSear
chRec);
while FindResult = 0 do
begin
if ((DSearchRec.Attr and faDirectory) = faDirectory) and not
IsDirNotation(DSearchRec.Name) then FindFiles(APath+DSearchRec.Name);
// Рекурсия
FindResult := FindNext(DSearchRec);
end;
finally
FindClose(FSearchRec);
end;
end;
procedure TMainForm.btnSearchForFilesClick(Sender: TObject);
{ Этот метод запускает процесс поиска. Сначала курсор прини
мает вид песочных часов, означающих, что для выполнения по
иска потребуется некоторое время. Затем очищается список и
рекурсивно вызывается функция FindFiles() для просмотра под
каталогов. }
begin
Screen.Cursor := crHourGlass;
 try
IbFiles.Items.Clear;
FFileName := edtFileMask.Text; FindFiles(dolDirectories.Directory);
Finally
Screen.Cursor := crDefault;
end;
end;
procedure TMainForm.dcbDrivesChange(Sender: TObject);
begin
dolDirectories. Drive := dcbDrives.Drive;
end;
end.
```

Перебор файлов с использованием C#.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.InteropServices;
using System.IO;
using FILETIME = System.Runtime.InteropServices.ComTypes. FILETIME;
using System.ComponentModel;
```



```
namespace FindFiles
{
    public class RsdnDirectory
    {
        /// <summary>
        /// Формирует путь требуемый функцией FindFirstFile.
        /// </summary>
        private static string MakePath(string path)
        {
            return Path.Combine(path, «*»);
        }

        /// <summary>
        /// Возвращает список файлов или каталогов
        ///находящихся по заданному пути path.
        /// </summary>
        /// <param name=«path»>Путь для которого нужно
        ///возвратить список.
</param>
        /// <param name=«isGetDirs»>
        /// Если true - функция возвращает список каталогов,
        ///иначе файлов.
        /// </param>
        /// <returns>Список файлов или каталогов.
</returns>
        private static IEnumerable<string>
        ///GetInternal(string path, bool isGetDirs)
        {
            // Структура в которую функции FindFirstFile и
            ///FindNextFile
            // возвращают информацию о текущем файле.
            WIN32_FIND_DATA findData;
            // Получаем информацию о текущем файле
            //и дескриптор перечислителя.
            // Этот дескриптор требуется передавать функции
FindNextFile для получения следующих файлов.
    IntPtr findHandle = FindFirstFile(MakePath(path), out findData);
            // Если произошла ошибка, то
            // нужно вынуть информацию об ошибке
            // и перепаковать ее в исключение.
if (findHandle == INVALID_HANDLE_VALUE)
throw new Win32Exception(Marshal.GetLastWin32Error());
            try
            {
```

```
            do
                if (isGetDirs ?
   (findData.dwFileAttributes & FileAttributes.Directory) != 0
: (findData.dwFileAttributes & FileAttributes.Directory) == 0)
                yield return findData.cFileName;
            while (FindNextFile(findHandle, out findData));
            }
            finally
            {
                FindClose(findHandle);
            }
    }

    /// <summary>
    /// Возвращает список файлов для некоторого пути.
    /// </summary>
    /// <param name=«path»>
    /// Каталог для которого нужно получить
    /// список файлов.
    /// </param>
    /// <returns>Список файлов каталога.</returns>
    public static IEnumerable<string> GetFilse(string path)
    {
        return GetInternal(path, false);
    }

    /// <summary>
    /// Возвращает список каталогов для некоторого пути.
    ///Функция не перебирает вложенные подкаталоги!
    /// </summary>
    /// <param name=«path»>
    /// Каталог для которого нужно получить список подкаталогов.
    /// </param>
    /// <returns>Список файлов каталога.</returns>
    public static IEnumerable<string> GetDirectories(string path)
    {
        return GetInternal(path, true);
    }

    /// <summary>
    /// Функция возвращает список относительных путей ко всем
    /// подкаталогам
    /// (в том числе и вложенным) заданного пути.
    /// </summary>
```



```
    /// <param name=«path»>Путь для которого унжно получить
    /// подкаталоги.</param>
    /// <returns>Список подкатлогов.
</returns> public static IEnumerable<string> GetAllDirecto-
ries(string path)
    {
    // Сначала перебираем подкаталоги первого уровня
    // вложенности...
    foreach (string subDir in GetDirectories(path))
    {
    // игнорируем имя текущего каталога и родительского.
      if (subDir == «..» || subDir == «.»)
        continue;

    // Комбинируем базовый путь и имя подкаталога.
    string relativePath = Path.Combine(path, subDir);

    // возвращаем пользователю относительный путь.
        yield return relativePath;

    // Создаем рекурсивно итератор для каждого подкаталога и...
    // возвращаем каждый его элемент в качестве элементов
    // текущего итератора.
    // Этот прием позвсляет обойти ограничение итераторов C# 2.0
    // связанное с невозможностью вызовов «yield return» из
    // функций вызваемых из
    // функции итератора. К сожалению это приводит к созданию
    // временного вложенного итератора на каждом шаге
    // рекурсии но затраты на создание такого объекта
    // относительно невелики, а удобство очень даже ощутимо.
    foreach (string subDir2 in GetAllDirectories(relativePath))
        yield return subDir2;
    }
}

    #region Импорт из kernel32

    private const int MAX_PATH = 260;

    [Serializable]
    [StructLayout(LayoutKind.Sequential, CharSet =
    CharSet.Auto)]
    [BestFitMapping(false)]
    private struct WIN32_FIND_DATA
```

```
    {
        public FileAttributes dwFileAttributes;
        public FILETIME ftCreationTime;
        public FILETIME ftLastAccessTime;
        public FILETIME ftLastWriteTime;
        public int nFileSizeHigh;
        public int nFileSizeLow;
        public int dwReserved0;
        public int dwReserved1;
        [MarshalAs(UnmanagedType.ByValTStr, SizeConst = MAX_PATH)]
        public string cFileName;
        [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 14)]
            public string cAlternate;
        }

        [DllImport(«kernel32», CharSet = CharSet.Auto,
// SetLastError = true)]
        private static extern IntPtr FindFirstFile(string
lpFileName,
        out WIN32_FIND_DATA lpFindFileData);

        [DllImport(«kernel32», CharSet = CharSet.Auto,
// SetLastError = true)]
        private static extern bool FindNextFile(IntPtr hFindFile,
            out WIN32_FIND_DATA lpFindFileData);

        [DllImport(«kernel32.dll», SetLastError = true)]
        private static extern bool FindClose(IntPtr hFindFile);

        private static readonly IntPtr INVALID_HANDLE_VALUE = new
IntPtr(-1);

        #endregion
    }
}
```

ЗАДАНИЯ ДЛЯ САМОСТОЯТЕЛЬНОГО ВЫПОЛНЕНИЯ

Указание. Главный поток обеспечивает реакцию формы. Вспомогательный поток запускает потоки для поиска файлов и ожидает от них результатов (использует для этого средства синхронизации).

1. Заданы два каталога. Сравнить, в каком из них больше вложенных каталогов. Вывести все.

2. Задано устройство (DriveComboBox). В первом окне (DirectoryListBox) отображается список всех каталогов выбранного устройства. При выделении каталога в первом окне вывести в первый ListBox список всех файлов этого каталога и статистику (количество файлов каждого типа: системных, скрытых и т.д.). Аналогично при выделении каталога в первом ListBoxe, во втором ListBoxe отображается информация о его содержимом. Так же для второго и третьего ListBoxa.

3. Задано устройство (DriveComboBox). В первом окне (DirectoryListBox) отображается список всех каталогов выбранного устройства. При выделении каталога в первом окне вывести в первый ListBox список всех вложенных каталогов этого каталога и информацию о количестве файлов, созданных ранее и позже заданной даты. Аналогично при выделении каталога в первом ListBoxe, во втором ListBoxe отображается информация о его содержимом. Так же для второго и третьего ListBoxa.

4. Заданы два каталога. Проверить, есть ли в них совпадающие (по названию). Вывести уникальные.

5. Заданы два каталога. Проверить, есть ли в них совпадающие (по названию). Вывести названия тех, для которых есть совпадения.

6. Заданы два каталога. Проверить, есть ли во втором файлы, созданные раньше, чем любой из первого каталога.

7. Заданы два каталога. Проверить, есть ли во втором файлы, созданные позже, чем любой из первого каталога.

8. Заданы два каталога. Сравнить их по количеству скрытых и только скрытых.

9. Заданы два каталога. Сравнить их по количеству Readonly и только Readonly.

10. Заданы два каталога. Сравнить их по количеству архивных и только архивных.

11. Заданы два каталога. Сравнить общий объем содержащихся в каждом из них файлов.

12. Заданы два каталога. Для каждого из них для всех вложенных каталогов вывести общее количество файлов, количество скрытых, архивных, только для чтения, системных, каталогов и прочих.

13. Заданы два каталога. Для каждого из них найти, вывести и сравнить общее количество файлов, диапазон дат создания.

14. Заданы два каталога. В каждом из них поменять у файлов, имеющих атрибут faReadonly, на faHidden.

15. Заданы два каталога. В каждом из них найти файлы, у которых время последнего доступа и время создания совпадают. Сравнить их количество.

16. Заданы два каталога. Для каждого из них найти файлы, у которых дата модификации позже заданной даты. Сравнить их количество.

17. Заданы два каталога. Для каждого подсчитать количество файлов, размер которых превышает заданное значение. Сравнить их количество.

18. Заданы два каталога. Для каждого из них подсчитать количество каталогов, содержащих более чем *m* файлов. Сравнить их количество.

19. Заданы два каталога. Для каждого из них подсчитать количество скрытых подкаталогов. Сравнить их количество.

20. Заданы два каталога. Для каждого из них подсчитать количество каталогов, не содержащих скрытых файлов. Сравнить их количество.

МОМЕНТАЛЬНЫЕ СНИМКИ

в Windows 95/98: использование ToolHelp32

ToolHelp32 — это семейство функций и процедур, составляющих подмножество Win32 API, которые позволяют получить сведения о некоторых низкоуровневых аспектах работы ОС. В частности, сюда входят функции, с помощью которых можно получить информацию обо всех процессах, выполняющихся в системе в данный момент, а также потоках, модулях и кучах, принадлежащих каждому процессу.

Типы и определения функций ToolHe1p32 размещаются в модуле TlHe1p32, поэтому при работе с этими функциями нужно включить его имя в список инструкции uses.

Благодаря многозадачной природе среды Win32 такие объекты, как процессы, потоки, модули и т.п., постоянно создаются, разрушаются и модифицируются. И поскольку состояние компьютера непрерывно изменяется, системная информация, которая, возможно, будет иметь значение в данный момент, через секунду уже никого не заинтересует. Например, предположим, что вы хотите написать программу для регистрации всех модулей, загруженных в систему. Поскольку операционная система в любое время может прервать выполнение потока, отрабатывающего вашу программу, чтобы предоставить какие-то кванты времени другому потоку в системе, модули теоретически могут создаваться и разрушаться даже в момент выборки информации о них.

В этой динамической среде имело бы смысл на мгновение заморозить систему, чтобы получить такую системную информацию. В ToolHelp32 не предусмотрено средств замораживания системы, но есть функция, с помощью которой можно сделать «снимок» системы в заданный момент времени. Эта функция называется CreateToolhelp32Snapshot(), и ее объявление выглядит следующим образом:

```
function CreateToolhelp32Snapshot(dwFlags, th32ProcessID:
DWORD): THandle;
stdcall;
```

Параметр dwFlags означает тип информации, подлежащий включению в моментальный снимок. Этот параметр может иметь одно из перечисленных в таблице значений.

Значение	Описание
TH32CS_INHERIT	Означает, что дескриптор снимка будет наследуемым
TH32CS_SNAPALL	Эквивалентно заданию значений TH32CS_SNAPHEAPLIST, TH32CS_SNAPMODULE, TH32CS_SNAPPROCESS и TH32CS_SNAPTHREAD
TH32CS_SNAPHEAPLIST	Включает в снимок список куч заданного процесса Win32
TH32CS_SNAPMODULE	Включает в снимок список модулей заданного процесса Win32
TH32CS_SNAPPROCESS	Включает в снимок список процессов Win32
TH32CS_SNAPTHREAD	Включает в снимок список потоков Win32

Функция CreateToolhelp32Snapshot() возвращает дескриптор созданного снимка или –1 в случае ошибки. Возвращаемый дескриптор работает подобно другим дескрипторам Win32 относительно процессов и потоков, для которых он действителен.

Следующий код создает дескриптор снимка, который содержит информацию обо всех процессах, загруженных в настоящий момент (EToolHelpError — это исключительная ситуация, определенная программистом):

```
var
Snap: THandle;
begin
Snap := CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
if Snap = -1 then
raise EToolHelpError.Create('CreateToolHelp32Snapshot failed');
end;
```

Для освобождения связанных с ним ресурсов используйте функцию `CloseHandle()`

Обработка информации о процессах

Имея дескриптор снимка, содержащий информацию о процессах, можно воспользоваться двумя функциями Tool1He1p32, которые позволяют последовательно просмотреть сведения обо всех процессах в системе. Функции Process32First() и Process32Next() определены следующим образом:

```
function Process32First(hSnapshot: THandle; var lppe:
TProcessEntry32):BOOL; stdcall;
function Process32Next(hSnapshot: THandle; var lppe:
TProcessEntry32): BOOL; stdcall;
```


Первый параметр у обеих функций является дескриптором снимка, возвращаемым функцией CreateToolhelp32Snapshot(). Второй параметр, lppe, представляет собой запись TProcessEntry32, которая передается по ссылке. По мере прохождения по элементам перечисления функции будут заполнять эту запись информацией о следующем процессе.

Запись TProcessEntry32 определяется так:

```
type
TProcessEntry32 = record
    dwSize: DWORD;
    cntUsage: DWORD;
    th32ProcessID: DWORD;
    th32DefaultHeapID: DWORD;
    th32ModuleID: DWORD;
    cntThreads: DWORD;
    th32ParentProcessID: DWORD;
    pcPriClassBase: Longint;
    dwFlags: DWORD;
    szExeFile: array[0..MAX_PATH - 1] of Char;
end;
```

- В поле dwSize содержится размер записи TProcessEntry32. До использования этой записи поле dwSize должно быть инициализировано значением SizeOf (TProcessEntry32).
- В поле cntUsage хранится значение счетчика ссылок процесса. Когда это значение станет равным нулю, операционная система выгрузит процесс.
- В поле th32ProcessID содержится идентификационный номер процесса.
- Поле th32DefaultHeapID предназначено для хранения идентификатора (ID) для кучи процесса, действующей по умолчанию. Этот ID имеет значение только для функций Tool1He1p32, и его нельзя использовать с другими функциями Win32.
- Поле thModuleID идентифицирует модуль, связанный с процессом. Это поле имеет значение только для функций ToolHelp32.
- По значению поля cntThreads можно судить о том, сколько потоков начало выполняться в данном процессе.
- Поле th32ParentProcessID идентифицирует родительский процесс для данного процесса.
- В поле pcPriClassBase хранится базовый приоритет процесса. Операционная система использует это значение для управления работой потоков.
- Поле dwFlags зарезервировано.
- В поле szExeFile содержится строка с ограничивающим нуль-символом, которая представляет собой путь и имя файла EXE-программы или драйвера, связанного с данным процессом.

После создания снимка, содержащего информацию о процессах, для опроса данных по каждому процессу следует вызвать сначала функцию Process32First(), а затем вызывать функцию Process32Next() до тех пор, пока она не вернет значение False.

```
procedure TMainForm.Button1Click(Sender: TObject);
var
  T: TProcessEntry32;
  FCurSnap:THandle;
  ListItem: TListItem;
begin
    T.dwSize := SizeOf(T);
    FCurSnap := CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    lvProcList.Items.Clear;
    with lvProcList do
    begin
    if Process32First(FCurSnap,T) then
    repeat
         ListItem := Items.Add;
         ListItem.Caption :=T.szExeFile;
         ListItem.SubItems.Add(IntToStr(T.cntThreads));
         ListItem.SubItems.Add(IntToStr(T.th32ProcessID));
         ListI em.SubItems.Add(IntToStr(T.th32ParentProcessID));
         ListItem.SubItems.Add(IntToStr(T.dwSize));
    until not Process32Next(FCurSnap, T);
    end;
end;
```

Обработка информации о потоках

Для составления списка потоков некоторого процесса в Tool Help32 предусмотрены две функции, которые аналогичны функциям, предназначенным для регистрации процессов: Thread32First() и Thread32Next(). Эти функции объявляются следующим образом:

```
function Thread32First(hSnapshot: THandle; var lpte: TThreadEntry32):
BOOL; stdcall;
function Thread32Next(hSnapshot: THandle; var lpte: TThreadENtry32):
BOOL; stdcall;
```

Помимо обычного параметра hSnapshot, этим функциям также передается по ссылке параметр типа TThreadEntry32. Как и в случае функций, работающих с процессами, каждая из них заполняет запись TThreadEntry32, объявление которой имеет вид

```
type
TThreadEntry32 = record
dwSize: DWORD;
```



```
    cntUsage: DWORD;
    th32ThreadID: DWORD;
    th320wnerProcessID: DWORD;
    tpBasePri: Longint;
    tpDeltaPri: Longint;
    dwFlags: DWORD;
end;
```

- Поле *dwSize* определяет размер записи, и поэтому оно должно быть инициализировано значением SizeQf (TThreadEntry32) до использования этой записи.
- В поле *cntUsage* содержится счетчик ссылок данного потока. При обнулении этого счетчика поток выгружается операционной системой.
- Поле *th32ThreadID* представляет собой идентификационный номер потока, который имеет значение только для функций ToolHelp32.
- В поле *th32OwnerProcessID* содержится идентификатор (ID) процесса, которому принадлежит данный поток. Этот ID можно использовать с другими функциями Win32.
- Поле *tpBasePri* представляет собой базовый класс приоритета потока. Это значение одинаково для всех потоков данного процесса. Возможные значения этого поля обычно лежат в диапазоне от 4 до 24. Описания этих значений приведены ниже.

Допустимые значения класса приоритета потоков

Значения	Описание
4	Ожидающий
8	Нормальный
13	Высокий
24	Реальное время

- Поле *tpDeltaPri* представляет собой дельта-приоритет (разницу), определяющий величину отличия реального приоритета от значения tpBasePri. Это число со знаком, которое в сочетании с базовым классом приоритета отображает общий приоритет потока.
- Поле *dwFlags* в данный момент зарезервировано и не должно использоваться.

Обработка информации о модулях

Опрос модулей выполняется практически так же, как опрос процессов или потоков. Для этого в Tool Help32 предусмотрены две функции: Module32First() и Module32Next(), которые определяются следующим образом:

```
function Module32First(hSnapshot: THandle; var lpme:
TModuleEntry32): BOOL; stdcall;
function Module32Next(hSnapshot: THandle; var lpme:
TModuleEntry32): BOOL; stdcall;
```

Первым параметром в обеих функциях является дескриптор снимка, а вторым var-параметром — запись TModuleEntry32. Ее определение имеет следующий вид:

```
type
TModuleEntry32 = record
   dwSize: DWORD;
   th32ModuleID: DWORD;
   th32ProcessID: DWORD;
   GlblcntUsage: DWORD;
   ProccntUsage: DWORD;
   modBaseAddr: PBYTE;
   modBaseSize: DWORD;
   hModule: HMODULE;
   szModule: array[0..MAX_MODULE_NAME32 + 1] of Char;
   szExePath: array[0..MAX_PATH - 1] of Char;
end;
```

- Поле *dwSize* определяет размер записи, поэтому должно быть инициализировано значением SizeOf (TModuleEntry32) до использования этой записи.
- Поле *th32ModuleID* представляет собой идентификатор модуля, который имеет значение только для функций ToolHelp32.
- Поле *th32ProcessID* содержит идентификатор (ID) опрашиваемого процесса. Этот ID можно использовать с другими функциями Win32.
- Поле *GlblcntUsage* содержит глобальный счетчик ссылок данного модуля.
- Поле *ProccntUsage* содержит счетчик ссылок модуля в контексте процесса-владельца.
- Поле *modBaseAddr* представляет собой базовый адрес модуля в памяти. Это значение действительно только в контексте идентификатора процесса th32ProcessID.
- Поле *modBaseSize* определяет размер (в байтах) модуля в памяти.
- В поле *hModule* содержится дескриптор модуля. Это значение действительно только в контексте идентификатора процесса th32ProcessID.
- В поле *szModule* содержится строка с именем модуля, завершающаяся нуль-символом.
- Поле *szExepath* предназначено для хранения строки с ограничивающим нуль-символом, содержащей полный путь модуля.

Обработка информации о динамической памяти (кучах)

Опрос куч несколько сложнее опроса других типов объектов. В Tool Help32 предусмотрены четыре функции, с помощью которых можно получить информацию о кучах. Первые две, Heap32ListFirst() и Heap32ListNext() позволяют выполнить проход по всем кучам процесса, а две другие, Heap32First() и Heap32Next(), используются для получения более подробной информации обо всех блоках внутри отдельной кучи.

Функции Heap32ListFirst() и Heap32ListNext() определяются следующим образом:

```
function Heap32ListFirst(hSnapshot: THandle; var lphl:
THeapList32): BOOL; stdcall;
function Heap32ListNext(hSnapshot: THandle; var lphl:
THeapList32): BOOL; stdcall;
```

И вновь первый параметр является дескриптором снимка, а второй lphl представляет собой запись типа THeapList32, передаваемую по ссылке. Определение этой записи имеет следующий вид:

```
type
THeapList32 = record
   dwSize: DWORD;
   th32ProcessID: DWORD;
   th32HeapID: DWORD;
   dwFlags: DWORD;
end;
```

- Поле *dwSize* определяет размер записи, и поэтому оно должно быть инициализировано значением SizeOf (THeapList32) до использования этой записи.
- В поле *th32ProcessID* содержится идентификатор (ID) процесса-владельца.
- В поле *th32HeapID* содержится идентификатор (ID) кучи. Этот ID имеет значение только для заданного процесса, и его можно использовать только с функциями Tool Help32.
- В поле *dwFlags* хранится признак, который определяет тип кучи. В качестве значения этого поля может использоваться либо константа HF32 DEFAULT (которая означает, что текущая куча является стандартной кучей процесса), либо константа HF32_SHARED (которая означает, что текущая куча является разделяемой обычным способом).

Функции Heap32First() и Heap32Next() определяются следующим образом:

```
function Heap32First(var lphe: THeapEntry32; th32ProcessID,
th32HeapID: DWORD): BOOL; stdcall;
function Heap32Next(var lphe: THeapEntry32): BOOL; stdcall;
```

Списки параметров этих функций немного отличаются от соответствующих списков функций, связанных с перечислением процессов, потоков, модулей. Эти функции предназначены для перечисления блоков данной кучи в данном процессе, а не некоторых свойств одного процесса, вызове функции Heap32First() параметры, установленные в полях th32ProcessID и th32HeapID, должны быть равны значениям одноименных полей записи THeapList32, заполненной с помощью функций Heap32ListFirst() или Heap32ListNext(). Var-параметр lphe функций Heap32First() и Heap32Next() имеет тип THeapEntry32. Эта запись содержит дескриптивную информацию, относящуюся к блоку кучи, и ее определение имеет следующий вид:

```
type
THeapEntry32=record
    dwSize: DWORD;
    hHandle: THandle;//дескриптор этого блока кучи
    dwAddress: DWORD;//линейный адрес начала блока
    dwBlockSize: DWORD;//размер блока в байтах
    dwFlags: DWORD;
    dwLockCount: DWORD;
    dwResvd: DWORD;
    th32ProcessID: DWORD; // Процесс-владелец
    th32HeapID: DWORD;// Идентификатор кучи в нем
end;
```

• Поле *dwSize* определяет размер записи и поэтому должно быть инициализировано значением SizeOf (THeapEntry32) до использования этой записи.

• В поле *hHandle* содержится дескриптор блока кучи.

• Поле *dwAddress* представляет собой линейный адрес начала блока кучи.

• В поле *dwBlockSize* содержится размер в байтах этого блока кучи.

• В поле *dwFlags* хранится признак, который определяет тип блока кучи. Это поле может иметь одно из значений:

LF32_FIXED – блок памяти имеет фиксированное местонахождение.

LF32_FREE – блок памяти не используется.

LF32 MOVEABLE – блок памяти можно перемещать.

• Поле *dwLockCount* представляет собой счетчик блокировок блока памяти. Это значение увеличивается на единицу при каждом вызове процессом функции GlobalLock() или LocalLock().

• Поле *dwResvd* зарезервировано в данный момент и не должно использоваться.

• В поле *th32ProcessID* содержится идентификатор процесса-владельца.

• Поле *th32HeapID* является идентификатором кучи, которой принадлежит блок.

Поскольку до составления списка блоков кучи вам придется сначала составить список куч, код опроса блоков кучи немного сложнее. В приведенном ниже методе WalkHeaps() вложены Heap32First()/Heap32Next() внутрь цикла Heap32ListFirst()/Heap32ListNext(). В этом методе вводится дополнительный уровень сложности путем добавления указателя на объекты типа записи PHeapEntry32 в ту часть массива DetailLists, которая относится к списку куч.

```
procedure WalkHeaps;
{ Использует функции ToolHelp32 для составления списка куч }
var
HL: THeapList32;
HE: THeapEntry32;
PHE: PHeapEntry32;
begin
DetailLists[ltHeap].Clear;
HL.dwSize := SizeOf(HL);
HE.dwSize:=SizeOf(HE);
If Heap32ListFirst(FCurSnap, HL) then
repeat
if Heap32First(HE, HL.th32ProcessID, HL.th32HeapID) then
repeat
{ Необходимо создать копию записи THeapList32, что позволит
сохранить информацию для просмотра сведений о куче позднее}
New(PHE);
PHE^ := HE;
DetailLists[ltHeap].AddObject(Format(SHeapStr,
[HL.th32HeapID, Pointer(HE.dwAddress), HE.dwBlockSize,
GetHeapFlagString(HE.dwFlags) ]), TObject(PHE));
until not Heap32Next(HE);
until not Heap32ListNext(FCurSnap, HL);
HeapListAlloc := True;
end;
```

ЗАДАНИЯ ДЛЯ САМОСТОЯТЕЛЬНОГО ВЫПОЛНЕНИЯ

1. Для каждого процесса, имеющего заданное число потоков, вывести идентификаторы этих потоков.

2. Из процессов с идентификаторами из заданного диапазона выбрать те, у которых модуль максимальной длины.

3. Выбрать процессы, которые находятся на заданном устройстве.

4. Выбрать процесс, у которого родительский процесс максимальной длины.

5. Выбрать процессы, у которых есть потоки с максимальным реальным приоритетом.

6. Вывести информацию о владельцах блоков, больших заданного *n*.

7. Вывести информацию о владельцах блоков из заданного диапазона линейных адресов.

8. Вывести информацию о владельцах самого большого объема свободной памяти.

9. Вывести информацию о владельцах самого большого объема фиксированной памяти.

10. Вывести информацию о владельцах самого большого числа блоков.

11. Вывести информацию о процессах, имеющих потоки с большим, чем они сами реальным приоритетом.

12. Вывести информацию о процессах, имеющих наибольшее число модулей.

13. Вывести информацию о процессах, имеющих наибольший суммарный размер модулей.

14. Вывести информацию о процессах, у которых есть потоки с большим, чем у них самих счетчиком ссылок.

15. Вывести информацию о процессах, у которых есть потоки с заданным значением счетчика ссылок.

16. Определить, есть ли модули, на которые ссылаются несколько процессов.

17. Для каждого процесса найти модуль наибольшей длины.

18. Определить, есть ли процесс с заданным количеством модулей.

19. Подсчитать количество процессов, у которых есть потоки с заданной величиной приоритета.

20. Для каждого процесса найти и вывести модуль максимального размера.

Алгоритмы планирования процессов

Планирование процессов включает в себя решение следующих задач:

1) определение момента времени для смены выполняемого процесса;

2) выбор процесса на выполнение из очереди готовых процессов;

3) переключение контекстов «старого» и «нового» процессов.

Первые две задачи решаются программными средствами, а последняя в значительной степени аппаратно.

Кроме этого, операционной системе для реализации планирования процессов требуется дополнительная информация: идентификатор процесса, состояние процесса, данные о степени привилегированности процесса, место нахождения кодового сегмента и другая информация. В некоторых ОС (например, в ОС UNIX) информацию такого рода, используемую ОС для планирования процессов, называют дескриптором процесса.

Существует множество различных алгоритмов планирования процессов, по-разному решающих вышеперечисленные задачи, преследующих различные цели и обеспечивающих различное качество мультипрограммирования. Среди этого множества алгоритмов рассмотрим подробнее две группы наиболее часто встречающихся алгоритмов: алгоритмы, основанные на квантовании, и алгоритмы, основанные на приоритетах.

В соответствии с алгоритмами, основанными на квантовании, смена активного процесса происходит, если:

- процесс завершился и покинул систему,
- произошла ошибка,
- процесс перешел в состояние ОЖИДАНИЕ,
- исчерпан квант процессорного времени, отведенный данному процессу.

Процесс, который исчерпал свой квант, переводится в состояние ГОТОВНОСТЬ и ожидает, когда ему будет предоставлен новый квант процессорного времени, а на выполнение в соответствии с определенным правилом выбирается новый процесс из очереди готовых. Таким образом, ни один процесс не занимает процессор надолго, поэтому квантование широко используется в системах разделения времени.

Кванты, выделяемые процессам, могут быть одинаковыми для всех процессов или различными. Кванты, выделяемые одному процессу, могут быть фиксированной величины или изменяться в разные периоды жизни процесса. Процессы, которые не полностью использовали выделенный им квант (например, из-за ухода на выполнение операций ввода-вывода), могут получить или не получить компенсацию в виде привилегий при последующем обслуживании. По разному может быть организована очередь готовых процессов: циклически, по правилу «первый пришел – первый обслужился» (FIFO) или по правилу «последний пришел – первый обслужился» (LIFO).

Другая группа алгоритмов использует понятие «приоритет» процесса. Приоритет – это число, характеризующее степень привилегированности процесса при использовании ресурсов вычислительной машины, в частности, процессорного времени: чем выше приоритет, тем выше привилегии.

Приоритет может выражаться целыми или дробными, положительным или отрицательным значением. Чем выше привилегии процесса, тем меньше времени он будет проводить в очередях. Приоритет может назначаться директивно администратором системы в зависимости от важности работы или внесенной платы, либо вычисляться самой ОС по определенным правилам, он может оставаться фиксированным на протяжении всей жизни процесса либо изменяться во времени в соответствии с некоторым законом. В последнем случае приоритеты называются динамическими.

Существуют две разновидности приоритетных алгоритмов: алгоритмы, использующие относительные приоритеты, и алгоритмы, использующие абсолютные приоритеты.

В обоих случаях выбор процесса на выполнение из очереди готовых осуществляется одинаково: выбирается процесс, имеющий наивысший приоритет. По разному решается проблема определения момента смены активного процесса. В системах с относительными приоритетами активный процесс выполняется до тех пор, пока он сам не покинет процессор, перейдя в состояние ОЖИДАНИЕ (или же произойдет ошибка, или процесс завершится). В системах с приоритетами выполнение активного процесса прерывается еще при одном условии: если в очереди готовых процессов появился процесс, приоритет которого выше приоритета активного процесса. В этом случае прерванный процесс переходит в состояние готовности.

Во многих операционных системах алгоритмы планирования построены с использованием как квантования, так и приоритетов. Например, в основе планирования лежит квантование, но величина кванта и/или порядок выбора процесса из очереди готовых определяется приоритетами процессов.

Вытесняющие и невытесняющие алгоритмы планирования

Существует два основных типа процедур планирования процессов – вытесняющие (preemptive) и невытесняющие (non-preemptive).

Non-preemptive multitasking – невытесняющая многозадачность – это способ планирования процессов, при котором активный процесс выполняется до тех пор, пока он сам, по собственной инициативе, не отдаст управление планировщику операционной системы для того, чтобы тот выбрал из очереди другой, готовый к выполнению процесс.

Preemptive multitasking – вытесняющая многозадачность – это такой способ, при котором решение о переключении процессора с выполнения одного процесса на выполнение другого процесса принимается планировщиком операционной системы, а не самой активной задачей.

Понятия preemptive и non-preemptive иногда отождествляются с понятиями приоритетных и бесприоритетных дисциплин, что совершенно не-

верно, а также с понятиями абсолютных и относительных приоритетов, что неверно отчасти. Вытесняющая и невытесняющая многозадачность – это более широкие понятия, чем типы приоритетности. Приоритеты задач могут как использоваться, так и не использоваться и при вытесняющих, и при невытесняющих способах планирования. Так в случае использования приоритетов дисциплина относительных приоритетов может быть отнесена к классу систем с невытесняющей многозадачностью, а дисциплина абсолютных приоритетов – к классу систем с вытесняющей многозадачностью. А бесприоритетная дисциплина планирования, основанная на выделении равных квантов времени для всех задач, относится к вытесняющим алгоритмам.

Основным различием между preemptive и non-preemptive вариантами многозадачности является степень централизации механизма планирования задач. При вытесняющей многозадачности механизм планирования задач целиком сосредоточен в операционной системе, и программист пишет свое приложение, не заботясь о том, что оно будет выполняться параллельно с другими задачами. При этом операционная система выполняет следующие функции: определяет момент снятия с выполнения активной задачи, запоминает ее контекст, выбирает из очереди готовых задач следующую и запускает ее на выполнение, загружая ее контекст.

При невытесняющей многозадачности механизм планирования распределен между системой и прикладными программами. Прикладная программа, получив управление от операционной системы, сама определяет момент завершения своей очередной итерации и передает управление ОС с помощью какого-либо системного вызова, а ОС формирует очереди задач и выбирает в соответствии с некоторым алгоритмом (например, с учетом приоритетов) следующую задачу на выполнение. Такой механизм создает проблемы, как для пользователей, так и для разработчиков.

Для пользователей это означает, что управление системой теряется на произвольный период времени, который определяется приложением (а не пользователем). Если приложение тратит слишком много времени на выполнение какой-либо работы, например, на форматирование диска, пользователь не может переключиться с этой задачи на другую задачу, например, на текстовый редактор, в то время как форматирование продолжалось бы в фоновом режиме. Эта ситуация нежелательна, так как пользователи обычно не хотят долго ждать, когда машина завершит свою задачу.

Поэтому разработчики приложений для non-preemptive операционной среды, возлагая на себя функции планировщика, должны создавать приложения так, чтобы они выполняли свои задачи небольшими частями. Например, программа форматирования может отформатировать одну дорожку дискеты и вернуть управление системе. После выполнения других задач система возвратит управление программе форматирования, чтобы та отформатировала следующую дорожку. Подобный метод разделения времени

между задачами работает, но он существенно затрудняет разработку программ и предъявляет повышенные требования к квалификации программиста. Программист должен обеспечить «дружественное» отношение своей программы к другим выполняемым одновременно с ней программам, достаточно часто отдавая им управление. Крайним проявлением «недружественности» приложения является его зависание, которое приводит к общему краху системы. В системах с вытесняющей многозадачностью такие ситуации, как правило, исключены, так как центральный планирующий механизм снимет зависшую задачу с выполнения.

Однако распределение функций планировщика между системой и приложениями не всегда является недостатком, а при определенных условиях может быть и преимуществом, потому, что дает возможность разработчику приложений самому проектировать алгоритм планирования, наиболее подходящий для данного фиксированного набора задач. Так как разработчик сам определяет в программе момент времени отдачи управления, то при этом исключаются нерациональные прерывания программ в «неудобные» для них моменты времени. Кроме того, легко разрешаются проблемы совместного использования данных: задача во время каждой итерации использует их монопольно и уверена, что на протяжении этого периода никто другой не изменит эти данные. Существенным преимуществом nonpreemptive систем является более высокая скорость переключения с задачи на задачу.

Примером эффективного использования невытесняющей многозадачности является файл-сервер NetWare, в котором, в значительной степени благодаря этому, достигнута высокая скорость выполнения файловых операций. Менее удачным оказалось использование невытесняющей многозадачности в операционной среде Windows 3.x.

Однако почти во всех современных операционных системах, ориентированных на высокопроизводительное выполнение приложений (UNIX, Windows NT, OS/2, VAX/VMS), реализована вытесняющая многозадачность. В последнее время дошла очередь и до ОС класса настольных систем, например, OS/2 Warp и Windows 95. Возможно в связи с этим вытесняющую многозадачность часто называют истинной многозадачностью.

Управление вводом-выводом

Одной из главных функций ОС является управление всеми устройствами ввода-вывода компьютера. ОС должна передавать устройствам команды, перехватывать прерывания и обрабатывать ошибки; она также должна обеспечивать интерфейс между устройствами и остальной частью системы. В целях развития интерфейс должен быть одинаковым для всех типов устройств (независимость от устройств).

Физическая организация устройств ввода-вывода

Устройства ввода-вывода делятся на два типа: блок-ориентированные устройства и байт-ориентированные устройства. Блок-ориентированные устройства хранят информацию в блоках фиксированного размера, каждый из которых имеет свой собственный адрес. Самое распространенное блок-ориентированное устройство – диск. Байт-ориентированные устройства не адресуемы и не позволяют производить операцию поиска, они генерируют или потребляют последовательность байтов. Примерами являются терминалы, строчные принтеры, сетевые адаптеры. Однако некоторые внешние устройства не относятся ни к одному классу, например, часы, которые, с одной стороны, не адресуемы, а с другой стороны, не порождают потока байтов. Это устройство только выдает сигнал прерывания в некоторые моменты времени.

Внешнее устройство обычно состоит из механического и электронного компонента. Электронный компонент называется контроллером устройства или адаптером. Механический компонент представляет собственно устройство. Некоторые контроллеры могут управлять несколькими устройствами. Если интерфейс между контроллером и устройством стандартизован, то независимые производители могут выпускать совместимые как контроллеры, так и устройства.

Операционная система обычно имеет дело не с устройством, а с контроллером. Контроллер, как правило, выполняет простые функции, например, преобразует поток бит в блоки, состоящие из байт, и осуществляют контроль и исправление ошибок. Каждый контроллер имеет несколько регистров, которые используются для взаимодействия с центральным процессором. В некоторых компьютерах эти регистры являются частью физического адресного пространства. В таких компьютерах нет специальных операций ввода-вывода. В других компьютерах адреса регистров ввода-вывода, называемых часто портами, образуют собственное адресное пространство за счет введения специальных операций ввода-вывода (например, команд IN и OUT в процессорах i86).

ОС выполняет ввод-вывод, записывая команды в регистры контроллера. Например, контроллер гибкого диска IBM PC принимает 15 команд, таких как READ, WRITE, SEEK, FORMAT и т.д. Когда команда принята, процессор оставляет контроллер и занимается другой работой. При завершении команды контроллер организует прерывание для того, чтобы передать управление процессором операционной системе, которая должна проверить результаты операции. Процессор получает результаты и статус устройства, читая информацию из регистров контроллера.

Организация программного обеспечения ввода-вывода

Основная идея организации программного обеспечения ввода-вывода состоит в разбиении его на несколько уровней, причем нижние уровни обеспечивают экранирование особенностей аппаратуры от верхних, а те, в свою очередь, обеспечивают удобный интерфейс для пользователей.

Ключевым принципом является независимость от устройств. Вид программы не должен зависеть от того, читает ли она данные с гибкого диска или с жесткого диска.

Очень близкой к идее независимости от устройств является идея единообразного именования, то есть для именования устройств должны быть приняты единые правила.

Другим важным вопросом для программного обеспечения ввода-вывода является обработка ошибок. Ошибки следует обрабатывать как можно ближе к аппаратуре. Если контроллер обнаруживает ошибку чтения, то он должен попытаться ее скорректировать. Если же это ему не удается, то исправлением ошибок должен заняться драйвер устройства. Многие ошибки могут исчезать при повторных попытках выполнения операций ввода-вывода, например, ошибки, вызванные наличием пылинок на головках чтения или на диске. И только если нижний уровень не может справиться с ошибкой, он сообщает об ошибке верхнему уровню.

Еще один ключевой вопрос – это использование блокирующих (синхронных) и неблокирующих (асинхронных) передач. Большинство операций физического ввода-вывода выполняется асинхронно – процессор начинает передачу и переходит на другую работу, пока не наступает прерывание. Пользовательские программы намного легче писать, если операции ввода-вывода блокирующие – после команды READ программа автоматически приостанавливается до тех пор, пока данные не попадут в буфер программы. ОС выполняет операции ввода-вывода асинхронно, но представляет их для пользовательских программ в синхронной форме.

Последняя проблема состоит в том, что одни устройства являются разделяемыми, а другие – выделенными. Диски – это разделяемые устройства, так как одновременный доступ нескольких пользователей к диску не представляет собой проблему. Принтеры – это выделенные устройства, потому что нельзя смешивать строчки, печатаемые различными пользователями. Наличие выделенных устройств создает для операционной системы некоторые проблемы.

Для решения поставленных проблем целесообразно разделить программное обеспечение ввода-вывода на четыре слоя:

- обработка прерываний,
- драйверы устройств,
- независимый от устройств слой операционной системы,
- пользовательский слой программного обеспечения.

Обработка прерываний

Прерывания должны быть скрыты как можно глубже в недрах операционной системы, чтобы как можно меньшая часть ОС имела с ними дело. Наилучший способ состоит в разрешении процессу, инициировавшему операцию ввода-вывода, блокировать себя до завершения операции и наступления прерывания. Процесс может блокировать себя, используя, например, вызов DOWN для семафора, или вызов WAIT для переменной условия, или вызов RECEIVE для ожидания сообщения. При наступлении прерывания процедура обработки прерывания выполняет разблокирование процесса, инициировавшего операцию ввода-вывода, используя вызовы UP, SIGNAL или посылая процессу сообщение. В любом случае эффект от прерывания будет состоять в том, что ранее заблокированный процесс теперь продолжит свое выполнение.

Драйверы устройств

Весь зависимый от устройства код помещается в драйвер устройства. Каждый драйвер управляет устройствами одного типа или, может быть, одного класса.

В операционной системе только драйвер устройства знает о конкретных особенностях какого-либо устройства. Например, только драйвер диска имеет дело с дорожками, секторами, цилиндрами, временем установления головки и другими факторами, обеспечивающими правильную работу диска.

Драйвер устройства принимает запрос от устройств программного слоя и решает, как его выполнить. Типичным запросом является чтение *n* блоков данных. Если драйвер был свободен во время поступления запроса, то он начинает выполнять запрос немедленно. Если же он был занят обслуживанием другого запроса, то вновь поступивший запрос присоединяется к очереди уже имеющихся запросов, и он будет выполнен, когда наступит его очередь.

Первый шаг в реализации запроса ввода-вывода, например, для диска, состоит в преобразовании его из абстрактной формы в конкретную. Для дискового драйвера это означает преобразование номеров блоков в номера цилиндров, головок, секторов, проверку, работает ли мотор, находится ли головка над нужным цилиндром. Короче говоря, он должен решить, какие операции контроллера нужно выполнить и в какой последовательности.

После передачи команды контроллеру драйвер должен решить, блокировать ли себя до окончания заданной операции или нет. Если операция занимает значительное время, как при печати некоторого блока данных, то драйвер блокируется до тех пор, пока операция не завершится, и обработчик прерывания не разблокирует его. Если команда ввода-вывода выполня-

ется быстро (например, прокрутка экрана), то драйвер ожидает ее завершения без блокирования.

Независимый от устройств слой операционной системы

Большая часть программного обеспечения ввода-вывода является независимой от устройств. Точная граница между драйверами и независимыми от устройств программами определяется системой, так как некоторые функции, которые могли бы быть реализованы независимым способом, в действительности выполнены в виде драйверов для повышения эффективности или по другим причинам.

Типичными функциями для независимого от устройств слоя являются:

- обеспечение общего интерфейса к драйверам устройств,
- именование устройств,
- защита устройств,
- обеспечение независимого размера блока,
- буферизация,
- распределение памяти на блок-ориентированных устройствах,
- распределение и освобождение выделенных устройств,
- уведомление об ошибках.

Остановимся на некоторых функциях данного перечня. Верхним слоям программного обеспечения неудобно работать с блоками разной величины, поэтому данный слой обеспечивает единый размер блока, например, за счет объединения нескольких различных блоков в единый логический блок. В связи с этим верхние уровни имеют дело с абстрактными устройствами, которые используют единый размер логического блока независимо от размера физического сектора.

При создании файла или заполнении его новыми данными необходимо выделить ему новые блоки. Для этого ОС должна вести список или битовую карту свободных блоков диска. На основании информации о наличии свободного места на диске может быть разработан алгоритм поиска свободного блока, независимый от устройства и реализуемый программным слоем, находящимся выше слоя драйверов.

Пользовательский слой программного обеспечения

Хотя большая часть программного обеспечения ввода-вывода находится внутри ОС, некоторая его часть содержится в библиотеках, связываемых с пользовательскими программами. Системные вызовы, включающие вызовы ввода-вывода, обычно делаются библиотечными процедурами. Если программа, написанная на языке С, содержит вызов count= write (fd, buffer,

nbytes), то библиотечная процедура write будет связана с программой. Набор подобных процедур является частью системы ввода-вывода. В частности, форматирование ввода или вывода выполняется библиотечными процедурами. Примером может служить функция printf языка С, которая принимает строку формата и, возможно, некоторые переменные в качестве входной информации, затем строит строку символов ASCII и делает вызов write для вывода этой строки. Стандартная библиотека ввода-вывода содержит большое число процедур, которые выполняют ввод-вывод и работают как часть пользовательской программы.

Другой категорией программного обеспечения ввода-вывода является подсистема спулинга (spooling). Спулинг – это способ работы с выделенными устройствами в мультипрограммной системе. Рассмотрим типичное устройство, требующее спулинга – строчный принтер. Хотя технически легко позволить каждому пользовательскому процессу открыть специальный файл, связанный с принтером, такой способ опасен из-за того, что пользовательский процесс может монополизировать принтер на произвольное время. Вместо этого создается специальный процесс – монитор, который получает исключительные права на использование этого устройства. Также создается специальный каталог, называемый каталогом спулинга. Для того чтобы напечатать файл, пользовательский процесс помещает выводимую информацию в этот файл и помещает его в каталог спулинга. Процесс-монитор по очереди распечатывает все файлы, содержащиеся в каталоге спулинга.

Управление распределёнными ресурсами

Базовые примитивы передачи сообщений в распределенных системах

Единственным по-настоящему важным отличием распределенных систем от централизованных является межпроцессная взаимосвязь. В централизованных системах связь между процессами, как правило, предполагает наличие разделяемой памяти. Типичный пример – проблема «поставщик-потребитель», в этом случае один процесс пишет в разделяемый буфер, а другой – читает из него. Даже наиболее простая форма синхронизации – семафор – требует, чтобы хотя бы одно слово (переменная самого семафора) было разделяемым. В распределенных системах нет какой бы то ни было разделяемой памяти, таким образом вся природа межпроцессных коммуникаций должна быть продумана заново.

Основой этого взаимодействия может служить только передача по сети сообщений. В самом простом случае системные средства обеспечения связи могут быть сведены к двум основным системным вызовам (примитивам), один – для посылки сообщения, другой – для получения сообщения. В

дальнейшем на их базе могут быть построены более мощные средства сетевых коммуникаций, такие как распределенная файловая система или вызов удаленных процедур, которые, в свою очередь, также могут служить основой для построения других сетевых сервисов.

Несмотря на концептуальную простоту этих системных вызовов – ПОСЛАТЬ и ПОЛУЧИТЬ – существуют различные варианты их реализации, от правильного выбора которых зависит эффективность работы сети. В частности, эффективность коммуникаций в сети зависит от способа задания адреса, от того, является ли системный вызов блокирующим или неблокирующим, какие выбраны способы буферизации сообщений и насколько надежным является протокол обмена сообщениями.

Способы адресации

Для того чтобы послать сообщение, необходимо указать адрес получателя. В очень простой сети адрес может задаваться в виде константы, но в более сложных сетях нужен и более изощренный способ адресации.

Одним из вариантов адресации на верхнем уровне является использование физических адресов сетевых адаптеров. Если в получающем компьютере выполняется только один процесс, то ядро будет знать, что делать с поступившим сообщением – передать его этому процессу. Однако, если на машине выполняется несколько процессов, то ядру не известно, какому из них предназначено сообщение, поэтому использование сетевого адреса адаптера в качестве адреса получателя приводит к очень серьезному ограничению – на каждой машине должен выполняться только один процесс.

Альтернативная адресная система использует имена назначения, состоящие из двух частей, определяющие номер машины и номер процесса. Однако адресация типа «машина-процесс» далека от идеала, в частности, она не гибка и не прозрачна, так как пользователь должен явно задавать адрес машины-получателя. В этом случае, если в один прекрасный день машина, на которой работает сервер, отказывает, то программа, в которой жестко используется адрес сервера, не сможет работать с другим сервером, установленном на другой машине.

Другим вариантом могло бы быть назначение каждому процессу уникального адреса, который никак не связан с адресом машины. Одним из способов достижения этой цели является использование централизованного механизма распределения адресов процессов, который работает просто, как счетчик. При получении запроса на выделение адреса он возвращает текущее значение счетчика, а затем наращивает его на единицу. Недостатком этой схемы является то, что централизованные компоненты, подобные этому, не обеспечивают в достаточной степени расширяемость систем. Еще один метод назначения процессам уникальных идентификаторов заключа-

ется в разрешении каждому процессу выбора своего собственного идентификатора из очень большого адресного пространства, такого как пространство 64-х битных целых чисел. Вероятность выбора одного и того же числа двумя процессами является ничтожной, а система хорошо расширяется. Однако здесь имеется одна проблема: как процесс-отправитель может узнать номер машины процесса-получателя. В сети, которая поддерживает широковещательный режим (то есть в ней предусмотрен такой адрес, который принимают все сетевые адаптеры), отправитель может широковещательно передать специальный пакет, который содержит идентификатор процесса назначения. Все ядра получат эти сообщения, проверят адрес процесса и, если он совпадает с идентификатором одного из процессов этой машины, пошлют ответное сообщение «Я здесь», содержащее сетевой адрес машины.

Хотя эта схема и прозрачна, но широковещательные сообщения перегружают систему. Такой перегрузки можно избежать, выделив в сети специальную машину для отображения высокоуровневых символьных имен. При применении такой системы процессы адресуются с помощью символьных строк, и в программы вставляются эти строки, а не номера машин или процессов. Каждый раз перед первой попыткой связаться, процесс должен послать запрос специальному отображающему процессу, обычно называемому сервером имен, запрашивая номер машины, на которой работает процесс-получатель.

Совершенно иной подход – это использование специальной аппаратуры. Пусть процессы выбирают свои адреса случайно, а конструкция сетевых адаптеров позволяет хранить эти адреса. Теперь адреса процессов не обнаруживаются путем широковещательной передачи, а непосредственно указываются в кадрах, заменяя там адреса сетевых адаптеров.

Блокирующие и неблокирующие примитивы

Примитивы бывают блокирующими и неблокирующими, иногда они называются соответственно синхронными и асинхронными. При использовании блокирующего примитива, процесс, выдавший запрос на его выполнение, приостанавливается до полного завершения примитива. Например, вызов примитива ПОЛУЧИТЬ приостанавливает вызывающий процесс до получения сообщения.

При использовании неблокирующего примитива управление возвращается вызывающему процессу немедленно, еще до того, как требуемая работа будет выполнена. Преимуществом этой схемы является параллельное выполнение вызывающего процесса и процесса передачи сообщения. Обычно в ОС имеется один из двух видов примитивов и очень редко - оба. Однако выигрыш в производительности при использовании неблокирующих примитивов компенсируется серьезным недостатком: отправитель не

может модифицировать буфер сообщения, пока сообщение не отправлено, а узнать, отправлено ли сообщение, отправитель не может. Отсюда сложности в построении программ, которые передают последовательность сообщений с помощью неблокирующих примитивов.

Имеется два возможных выхода. Первое решение – это заставить ядро копировать сообщение в свой внутренний буфер, а затем разрешить процессу продолжить выполнение. С точки зрения процесса эта схема ничем не отличается от схемы блокирующего вызова: как только процесс снова получает управление, он может повторно использовать буфер.

Второе решение заключается в прерывании процесса-отправителя после отправки сообщения, чтобы проинформировать его, что буфер снова доступен. Здесь не требуется копирование, что экономит время, но прерывание пользовательского уровня делает программирование запутанным, сложным, может привести к возникновению гонок.

Вопросом, тесно связанным с блокирующими и неблокирующими вызовами, является вопрос тайм-аутов. В системе с блокирующим вызовом ПОСЛАТЬ при отсутствии ответа вызывающий процесс может заблокироваться навсегда. Для предотвращения такой ситуации в некоторых системах вызывающий процесс может задать временной интервал, в течение которого он ждет ответ. Если за это время сообщение не поступает, вызов ПОСЛАТЬ завершается с кодом ошибки.

Буферизуемые и небуферизуемые примитивы

Примитивы, которые были описаны, являются небуферизуемыми примитивами. Это означает, что вызов ПОЛУЧИТЬ сообщает ядру машины, на которой он выполняется, адрес буфера, в который следует поместить пребывающее для него сообщение.

Эта схема работает прекрасно при условии, что получатель выполняет вызов ПОЛУЧИТЬ раньше, чем отправитель выполняет вызов ПОСЛАТЬ. Вызов ПОЛУЧИТЬ сообщает ядру машины, на которой выполняется, по какому адресу должно поступить ожидаемое сообщение, и в какую область памяти необходимо его поместить. Проблема возникает тогда, когда вызов ПОСЛАТЬ сделан раньше вызова ПОЛУЧИТЬ. Каким образом сможет узнать ядро на машине получателя, какому процессу адресовано вновь поступившее сообщение, если их несколько? И как оно узнает, куда его скопировать?

Один из вариантов – просто отказаться от сообщения, позволить отправителю взять тайм-аут и надеяться, что получатель все-таки выполнит вызов ПОЛУЧИТЬ перед повторной передачей сообщения. Этот подход не сложен в реализации, но, к сожалению, отправитель (или скорее ядро его машины) может сделать несколько таких безуспешных попыток. Еще хуже то, что после достаточно большого числа безуспешных попыток ядро отправителя может сделать неправильный вывод об аварии на машине получателя или о неправильности использованного адреса.

Второй подход к этой проблеме заключается в том, чтобы хранить хотя бы некоторое время, поступающие сообщения в ядре получателя на тот случай, что вскоре будет выполнен соответствующий вызов ПОЛУЧИТЬ. Каждый раз, когда поступает такое «неожидаемое» сообщение, включается таймер. Если заданный временной интервал истекает раньше, чем происходит соответствующий вызов ПОЛУЧИТЬ, то сообщение теряется.

Хотя этот метод и уменьшает вероятность потери сообщений, он порождает проблему хранения и управления преждевременно поступившими сообщениями. Необходимы буферы, которые следует где-то размещать, освобождать, в общем, которыми нужно управлять. Концептуально простым способом управления буферами является определение новой структуры данных, называемой почтовым ящиком.

Процесс, который заинтересован в получении сообщений, обращается к ядру с запросом о создании для него почтового ящика и сообщает адрес, по которому ему могут поступать сетевые пакеты, после чего все сообщения с данным адресом будут помещены в его почтовый ящик. Такой способ часто называют буферизуемым примитивом.

Надежные и ненадежные примитивы

Ранее подразумевалось, что когда отправитель посылает сообщение, адресат его обязательно получает. Но реально сообщения могут теряться. Предположим, что используются блокирующие примитивы. Когда отправитель посылает сообщение, то он приостанавливает свою работу до тех пор, пока сообщение не будет послано. Однако нет никаких гарантий, что после того, как он возобновит свою работу, сообщение будет доставлено адресату.

Для решения этой проблемы существует три подхода. Первый заключается в том, что система не берет на себя никаких обязательств по поводу доставки сообщений. Реализация надежного взаимодействия становится целиком заботой пользователя.

Второй подход заключается в том, что ядро принимающей машины посылает квитанцию-подтверждение ядру отправляющей машины на каждое сообщение. Посылающее ядро разблокирует пользовательский процесс только после получения этого подтверждения. Подтверждение передается от ядра к ядру. Ни отправитель, ни получатель его не видят.

Третий подход заключается в использовании ответа в качестве подтверждения в тех системах, в которых запрос всегда сопровождается ответом. Отправитель остается заблокированным до получения ответа. Если ответа нет слишком долго, то посылающее ядро может переслать запрос специальной службе предотвращения потери сообщений.

Вызов удаленных процедур (RPC)

Концепция удаленного вызова процедур

Идея вызова удаленных процедур (Remote Procedure Call – RPC) состоит в расширении хорошо известного и понятного механизма передачи управления и данных внутри программы, выполняющейся на одной машине, на передачу управления и данных через сеть. Средства удаленного вызова процедур предназначены для облегчения организации распределенных вычислений. Наибольшая эффективность использования RPC достигается в тех приложениях, в которых существует интерактивная связь между удаленными компонентами с небольшим временем ответов и относительно малым количеством передаваемых данных. Такие приложения называются RPC-ориентированными.

Характерными чертами вызова локальных процедур являются:

- асимметричность, то есть одна из взаимодействующих сторон является инициатором;
- синхронность, то есть выполнение вызывающей процедуры приостанавливается с момента выдачи запроса и возобновляется только после возврата из вызываемой процедуры.

Реализация удаленных вызовов существенно сложнее реализации вызовов локальных процедур. Начнем с того, что поскольку вызывающая и вызываемая процедуры выполняются на разных машинах, то они имеют разные адресные пространства, и это создает проблемы при передаче параметров и результатов, особенно если машины не идентичны. Так как RPC не может рассчитывать на разделяемую память, то это означает, что параметры RPC не должны содержать указателей на ячейки нестековой памяти, и что значения параметров должны копироваться с одного компьютера на другой. Следующим отличием RPC от локального вызова является то, что он обязательно использует нижележащую систему связи, однако это не должно быть явно видно ни в определении процедур, ни в самих процедурах. Удаленность вносит дополнительные проблемы. Выполнение вызывающей программы и вызываемой локальной процедуры в одной машине реализуется в рамках единого процесса. Но в реализации RPC участвуют как минимум два процесса -- по одному в каждой машине. В случае, если один из них аварийно завершится, могут возникнуть следующие ситуации: при аварии вызывающей процедуры удаленно вызванные процедуры станут «осиротевшими», а при аварийном завершении удаленных процедур станут «обездоленными родителями» вызывающие процедуры, которые будут безрезультатно ожидать ответа от удаленных процедур.

Кроме того, существует ряд проблем, связанных с неоднородностью языков программирования и операционных сред: структуры данных и структуры

вызова процедур, поддерживаемые в каком-либо одном языке программирования, не поддерживаются точно так же во всех других языках.

Эти и некоторые другие проблемы решает широко распространенная технология RPC, лежащая в основе многих распределенных операционных систем.

Базовые операции RPC

Чтобы понять работу RPC, рассмотрим вначале выполнение вызова локальной процедуры в обычной машине, работающей автономно. Пусть это, например, будет системный вызов

```
count=read (fd,buf,nbytes);
```

где fd – целое число,
buf – массив символов,
nbytes – целое число.

Чтобы осуществить вызов, вызывающая процедура помещает параметры в стек в обратном порядке. После того, как вызов read выполнен, он помещает возвращаемое значение в регистр, перемещает адрес возврата и возвращает управление вызывающей процедуре, которая выбирает параметры из стека, возвращая его в исходное состояние. Заметим, что в языке С параметры могут вызываться или по ссылке (by name), или по значению (by value). По отношению к вызываемой процедуре параметры-значения являются инициализируемыми локальными переменными. Вызываемая процедура может изменить их, и это не повлияет на значение оригиналов этих переменных в вызывающей процедуре.

Если в вызываемую процедуру передается указатель на переменную, то изменение значения этой переменной вызываемой процедурой влечет изменение значения этой переменной и для вызывающей процедуры. Этот факт весьма существенен для RPC.

Существует также другой механизм передачи параметров, который не используется в языке С. Он называется call-by-copy/restore и состоит в необходимости копирования вызывающей программой переменных в стек в виде значений, а затем копирования назад после выполнения вызова поверх оригинальных значений вызывающей процедуры.

Решение о том, какой механизм передачи параметров использовать, принимается разработчиками языка. Иногда это зависит от типа передаваемых данных. В языке С, например, целые и другие скалярные данные всегда передаются по значению, а массивы – по ссылке.

Идея, положенная в основу RPC, состоит в том, чтобы сделать вызов удаленной процедуры, выглядящим по возможности так же, как и вызов локальной процедуры. Другими словами – сделать RPC прозрачным: вызывающей процедуре не требуется знать, что вызываемая процедура находится на другой машине, и наоборот.

RPC достигает прозрачности следующим путем. Когда вызываемая процедура действительно является удаленной, в библиотеку помещается вместо локальной процедуры другая версия процедуры, называемая клиентским стабом (stub – заглушка). Подобно оригинальной процедуре, стаб вызывается с использованием вызывающей последовательности, так же происходит прерывание при обращении к ядру. Только в отличие от оригинальной процедуры он не помещает параметры в регистры и не запрашивает у ядра данные, вместо этого он формирует сообщение для отправки ядру удаленной машины.

Этапы выполнения RPC

Взаимодействие программных компонентов при выполнении удаленного вызова осуществляется следующим образом (рис. 3).

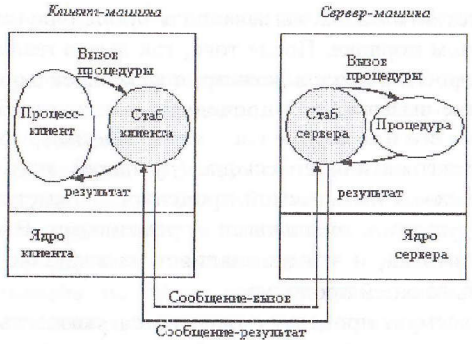


Рис. 3. Вызов удаленных процедур

• После того, как клиентский стаб был вызван программой-клиентом, его первой задачей является заполнение буфера отправляемым сообщением. В некоторых системах клиентский стаб имеет единственный буфер фиксированной длины, заполняемый каждый раз с самого начала при поступлении каждого нового запроса. В других системах буфер сообщения представляет собой пул буферов для отдельных полей сообщения, причем некоторые из этих буферов уже заполнены. Этот метод особенно подходит для тех случаев, когда пакет имеет формат, состоящий из большого числа полей, но значения многих из этих полей не меняются от вызова к вызову.

• Затем параметры должны быть преобразованы в соответствующий формат и вставлены в буфер сообщения. К этому моменту сообщение готово к передаче, поэтому выполняется прерывание по вызову ядра.

• Когда ядро получает управление, оно переключает контексты, сохраняет регистры процессора и карту памяти (дескрипторы страниц), устанавливает новую карту памяти, которая будет использоваться для работы в режиме ядра. Поскольку контексты ядра и пользователя различаются, ядро должно точно скопировать сообщение в свое собственное адресное пространство, так, чтобы иметь к нему доступ, запомнить адрес назначения (а, возможно, и другие поля заголовка), а также оно должно передать его сетевому интерфейсу. На этом завершается работа на клиентской стороне. Включается таймер передачи, и ядро может либо выполнять циклический опрос наличия ответа, либо передать управление планировщику, который выберет какой-либо другой процесс на выполнение. В первом случае ускоряется выполнение запроса, но отсутствует мультипрограммирование.

• На стороне сервера поступающие биты помещаются принимающей аппаратурой либо во встроенный буфер, либо в оперативную память. Когда вся информация будет получена, генерируется прерывание. Обработчик прерывания проверяет правильность данных пакета и определяет, какому стабу следует их передать. Если ни один из стабов не ожидает этот пакет, обработчик должен либо поместить его в буфер, либо вообще отказаться от него. Если имеется ожидающий стаб, то сообщение копируется ему. Наконец, выполняется переключение контекстов, в результате чего восстанавливаются регистры и карта памяти, принимая те значения, которые они имели в момент, когда стаб сделал вызов receive.

• Теперь начинает работу серверный стаб. Он распаковывает параметры и помещает их соответствующим образом в стек. Когда все готово, выполняется вызов сервера.

• После выполнения процедуры сервер передает результаты клиенту. Для этого выполняются все описанные выше этапы, только в обратном порядке.

Динамическое связывание

Рассмотрим вопрос о том, как клиент задает месторасположение сервера. Одним из методов решения этой проблемы является непосредственное использование сетевого адреса сервера в клиентской программе. Недостаток такого подхода - его чрезвычайная негибкость: при перемещении сервера, или при увеличении числа серверов, или при изменении интерфейса во всех этих и многих других случаях необходимо перекомпилировать все программы, которые использовали жесткое задание адреса сервера. Для того, чтобы избежать всех этих проблем, в некоторых распределенных системах используется так называемое динамическое связывание.

Начальным моментом для динамического связывания является формальное определение (спецификация) сервера. Спецификация содержит имя файл-сервера, номер версии и список процедур-услуг, предоставляемых

данным сервером для клиентов. Для каждой процедуры дается описание ее параметров с указанием того, является ли данный параметр входным или выходным относительно сервера. Некоторые параметры могут быть одновременно входными и выходными – например, некоторый массив, который посылается клиентом на сервер, модифицируется там, а затем возвращается обратно клиенту (операция copy/ restore).

Формальная спецификация сервера используется в качестве исходных данных для программы-генератора стабов, которая создает как клиентские, так и серверные стабы. Затем они помещаются в соответствующие библиотеки. Когда пользовательская (клиентская) программа вызывает любую процедуру, определенную в спецификации сервера, соответствующая стаб-процедура связывается с двоичным кодом программы. Аналогично, когда компилируется сервер, с ним связываются серверные стабы.

При запуске сервера самым первым его действием является передача своего серверного интерфейса специальной программе, называемой binder'ом. Этот процесс, известный как процесс регистрации сервера, включает передачу сервером своего имени, номера версии, уникального идентификатора и описателя местонахождения сервера. Описатель системно независим и может представлять собой IP, Ethernet, X.500 или еще какой-либо адрес. Кроме того, он может содержать и другую информацию, например, относящуюся к аутентификации.

Когда клиент вызывает одну из удаленных процедур первый раз, например, read, клиентский стаб видит, что он еще не подсоединен к серверу, и посылает сообщение binder-программе с просьбой об импорте интерфейса нужной версии нужного сервера. Если такой сервер существует, то binder передает описатель и уникальный идентификатор клиентскому стабу.

Клиентский стаб при посылке сообщения с запросом использует в качестве адреса описатель. В сообщении содержатся параметры и уникальный идентификатор, который ядро сервера использует для того, чтобы направить поступившее сообщение в нужный сервер в случае, если их несколько на этой машине.

Этот метод, заключающийся в импорте/экспорте интерфейсов, обладает высокой гибкостью. Например, может быть несколько серверов, поддерживающих один и тот же интерфейс, и клиенты распределяются по серверам случайным образом. В рамках этого метода становится возможным периодический опрос серверов, анализ их работоспособности и, в случае отказа, автоматическое отключение, что повышает общую отказоустойчивость системы. Этот метод может также поддерживать аутентификацию клиента. Например, сервер может определить, что он может быть использован только клиентами из определенного списка.

Однако у динамического связывания имеются недостатки, например, дополнительные накладные расходы (временные затраты) на экспорт и им-

порт интерфейсов. Величина этих затрат может быть значительна, так как многие клиентские процессы существуют короткое время, а при каждом старте процесса процедура импорта интерфейса должна быть снова выполнена. Кроме того, в больших распределенных системах может стать узким местом программа binder, а создание нескольких программ аналогичного назначения также увеличивает накладные расходы на создание и синхронизацию процессов.

Семантика RPC в случае отказов

В идеале RPC должен функционировать правильно и в случае отказов. Рассмотрим следующие классы отказов:

1. Клиент не может определить местонахождения сервера, например, в случае отказа нужного сервера, или из-за того, что программа клиента была скомпилирована давно и использовала старую версию интерфейса сервера. В этом случае в ответ на запрос клиента поступает сообщение, содержащее код ошибки.

2. Потерян запрос от клиента к серверу. Самое простое решение – через определенное время повторить запрос.

3. Потеряно ответное сообщение от сервера клиенту. Этот вариант сложнее предыдущего, так как некоторые процедуры не являются идемпотентными. Идемпотентной называется процедура, запрос на выполнение которой можно повторить несколько раз, и результат при этом не изменится. Примером такой процедуры может служить чтение файла. Но вот процедура снятия некоторой суммы с банковского счета не является идемпотентной, и в случае потери ответа повторный запрос может существенно изменить состояние счета клиента. Одним из возможных решений является приведение всех процедур к идемпотентному виду. Однако на практике это не всегда удается, поэтому может быть использован другой метод – последовательная нумерация всех запросов клиентским ядром. Ядро сервера запоминает номер самого последнего запроса от каждого из клиентов, и при получении каждого запроса выполняет анализ – является ли этот запрос первичным или повторным.

4. Сервер потерпел аварию после получения запроса. Здесь также важно свойство идемпотентности, но к сожалению не может быть применен подход с нумерацией запросов. В данном случае имеет значение, когда произошел отказ – до или после выполнения операции. Но клиентское ядро не может распознать эти ситуации, для него известно только то, что время ответа истекло. Существует три подхода к этой проблеме:

• Ждать до тех пор, пока сервер не перезагрузится и пытаться выполнить операцию снова. Этот подход гарантирует, что RPC был выполнен до конца по крайней мере один раз, а возможно и более.

• Сразу сообщить приложению об ошибке. Этот подход гарантирует, что RPC был выполнен не более одного раза.

• Третий подход не гарантирует ничего. Когда сервер отказывает, клиенту не оказывается никакой поддержки. RPC может быть или не выполнен вообще, или выполнен много раз. Во всяком случае этот способ очень легко реализовать.

Ни один из этих подходов не является очень привлекательным. А идеальный вариант, который бы гарантировал ровно одно выполнение RPC, в общем случае не может быть реализован по принципиальным соображениям. Пусть, например, удаленной операцией является печать некоторого текста, которая включает загрузку буфера принтера и установку одного бита в некотором управляющем регистре принтера, в результате которой принтер стартует. Авария сервера может произойти как за микросекунду до, так и за микросекунду после установки управляющего бита. Момент сбоя целиком определяет процедуру восстановления, но клиент о моменте сбоя узнать не может. Короче говоря, возможность аварии сервера радикально меняет природу RPC и ясно отражает разницу между централизованной и распределенной системой. В первом случае крах сервера ведет к краху клиента, и восстановление невозможно. Во втором случае действия по восстановлению системы выполнить и возможно, и необходимо.

5. Клиент потерпел аварию после отсылки запроса. В этом случае выполняются вычисления результатов, которых никто не ожидает. Такие вычисления называют «сиротами». Наличие сирот может вызвать различные проблемы: непроизводительные затраты процессорного времени, блокирование ресурсов, подмена ответа на текущий запрос ответом на запрос, который был выдан клиентской машиной еще до перезапуска системы.

Как поступать с сиротами? Рассмотрим 4 возможных решения.

• Уничтожение. До того, как клиентский стаб посылает RPC-сообщение, он делает отметку в журнале, оповещая о том, что он будет сейчас делать. Журнал хранится на диске или в другой памяти, устойчивой к сбоям. После аварии система перезагружается, журнал анализируется и сироты ликвидируются. К недостаткам такого подхода относятся, во-первых, повышенные затраты, связанные с записью о каждом RPC на диск, а, во-вторых, возможная неэффективность из-за появления сирот второго поколения, порожденных RPC-вызовами, выданными сиротами первого поколения.

• Перевоплощение. В этом случае все проблемы решаются без использования записи на диск. Метод состоит в делении времени на последовательно пронумерованные периоды. Когда клиент перезагружается, он передает широковещательное сообщение всем машинам о начале нового периода. После приема этого сообщения все удаленные вычисления ликвидируются. Конечно, если сеть сегментированная, то некоторые сироты могут и уцелеть.

• Мягкое перевоплощение аналогично предыдущему случаю, за исключением того, что отыскиваются и уничтожаются не все удаленные вычисления, а только вычисления перезагружающегося клиента.

• Истечение срока. Каждому запросу отводится стандартный отрезок времени T, в течение которого он должен быть выполнен. Если запрос не выполняется за отведенное время, то выделяется дополнительный квант. Хотя это и требует дополнительной работы, но если после аварии клиента сервер ждет в течение интервала T до перезагрузки клиента, то все сироты обязательно уничтожаются.

На практике ни один из этих подходов не желателен, более того, уничтожение сирот может усугубить ситуацию. Например, пусть сирота заблокировал один или более файлов базы данных. Если сирота будет вдруг уничтожен, то эти блокировки останутся, кроме того уничтоженные сироты могут остаться стоять в различных системных очередях, в будущем они могут вызвать выполнение новых процессов и т.п.

Список литературы

1. Олифер В. Г. Сетевые операционные системы : учебник для вузов / В. Г. Олифер; Н. А. Олифер. – 2-е изд. – СПб. [и др.] : Питер, 2008. – 668 с.

2. Таненбаум Э. Современные операционные системы / Э. Таненбаум. – 2-е изд. – СПб. : Питер, 2002. – 1037 с.

3. Гордеев А. В. Системное программное обеспечение : учебник для студ. вузов, обуч. по специальностям: «Вычисл. машины, комплексы, системы и сети» и «Автоматизир. системы обраб. информ. и упр.» направления подгот. дипломир. специалистов «Информ. и вычисл. техника» / А. В. Гордеев, А. Ю. Молчанов. – СПб. и др. : Питер, 2002. – 734 с.

4. Системное и прикладное программное обеспечение : учебно-методическое пособие / Воронеж. гос. ун-т; сост. : М.А. Артемов [и др.]. – Воронеж : ЛОП ВГУ, 2006. – 74 с.

Учебное издание

Вощинская Гильда Эдгаровна,
Артемов Михаил Анатольевич

ОПЕРАЦИОННЫЕ СИСТЕМЫ

Часть 2

Учебно-методическое пособие для вузов

Редактор И.Г. Валынкина

Компьютерная верстка О.В. Шкуратько

Подп. в печ. 21.11.2012. Формат 60×84/16.
Усл. печ. л. 3,8. Тираж 50 экз. Заказ 729.

Издательско-полиграфический центр
Воронежского государственного университета.
394000, г. Воронеж, пл. им. Ленина, 10. Тел. (факс): +7 (473) 259-80-26
http://www.ppc.vsu.ru; e-mail: pp_center@ppc.vsu.ru

Отпечатано с готового оригинал-макета
в типографии Издательско-полиграфического центра
Воронежского государственного университета.
394000, г. Воронеж, ул. Пушкинская, 3. Тел. +7 (473) 220-41-33