МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «ЛИПЕЦКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Институт компьютерных наук
Кафедра автоматизированных систем управления

Индивидуальное домашнее задание

по операционным системам

Студент АС-21-1 _____ Станиславчук С. М.

(подпись, дата)

Руководитель

Доцент, к.п.н. _____ Кургасов В. В.

(подпись, дата)

Липецк 2024

Содержание

2. Задание
- составить конспект;
- проиллюстрировать примерами;
- привести (по возможности) графическое пояснение (таблицы, графики, картинки, схемы);
- сделать выводы о проделанной работе.

Выбранная тема: “Глава 3: Процессы” (106-156)

3. Конспект

Глава 3. Процессы

Ранние компьютеры позволяли одновременно выполнять только одну программу.
Эта программа имела полный контроль над системой и имела доступ ко всем ее ресурсам.
Напротив, современные компьютерные системы позволяют загружать в память и выполнять одновременно несколько программ. Эта эволюция потребовала более жесткого контроля и большей разделения различных программ; и эти потребности привели к появлению понятия процесса, который представляет собой исполняемую программу.
Процесс — это единица работы в современной вычислительной системе. Чем сложнее операционная система, тем больше от нее ожидают действий от имени своих пользователей. Хотя его основной задачей является выполнение пользовательских программ, он также должен заботиться о различных системных задачах, которые лучше всего выполнять в пространстве пользователя, а не внутри ядра.
Таким образом, система состоит из набора процессов, некоторые из которых выполняют пользовательский код, другие — код операционной системы.
Потенциально все эти процессы могут выполняться одновременно с мультиплексированием ЦП (или ЦП). В этой главе вы прочтете о том, что такое процессы, как они представлены в операционной системе и как они работают.

ЦЕЛИ ГЛАВЫ
• Определить отдельные компоненты процесса и проиллюстрировать, как они представлены и запланированы в операционной системе.
• Описать, как процессы создаются и завершаются в операционной системе, включая разработку программ, использующих соответствующие системные вызовы, выполняющие эти операции.
• Описать и сравнить межпроцессное взаимодействие с использованием общей памяти и передачи сообщений.
• Разрабатывать программы, использующие каналы и общую память POSIX для межпроцессного взаимодействия.
• Описать взаимодействие клиент-сервер с использованием сокетов и удаленных вызовов процедур.
• Проектировать модули ядра, взаимодействующие с операционной системой Linux.

3.1 Концепция процесса

Вопрос, который возникает при обсуждении операционных систем, заключается в том, как называть всю деятельность ЦП.
Первые компьютеры представляли собой пакетные системы, выполнявшие задания, после чего появились системы с разделением времени, которые запускали пользовательские программы или задачи. Даже в однопользовательской системе пользователь может иметь возможность одновременно запускать несколько программ: текстовый процессор, веб-браузер и пакет электронной почты. И даже если компьютер может выполнять только одну программу одновременно, например, на встроенном устройстве, не поддерживающем многозадачность, операционной системе может потребоваться поддержка собственных внутренних программных действий, таких как управление памятью. Во многом все эти виды деятельности схожи, поэтому мы называем их всеми процессами. Хотя мы лично предпочитаем более современный термин «процесс», термин «задание» имеет историческое значение, поскольку большая часть теории и

терминологии операционных систем была разработана в то время, когда основным видом деятельности операционных систем была обработка заданий.
Поэтому в некоторых случаях мы используем job при описании роли операционной системы. Например, было бы ошибочно избегать использования общепринятых терминов, включающих слово «работа» (например, планирование работы), просто потому, что процесс заменил работу.

3.1.1 Процесс

Неформально, как упоминалось ранее, процесс — это исполняемая программа.
Статус текущей активности процесса представлен значением программного счетчика и содержимым регистров процессора. Структура памяти процесса обычно делится на несколько разделов и показана на рис. 3.1.
Эти разделы включают в себя:
• Текстовый раздел — исполняемый код.
• Раздел данных — глобальные переменные.

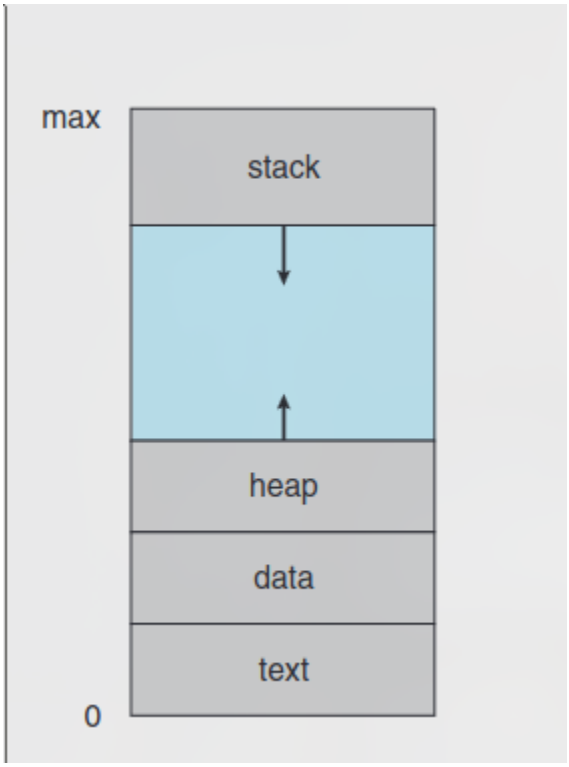


Рисунок 3.1 Расположение процесса в памяти

Обратите внимание, что размеры разделов текста и данных фиксированы, поскольку их размеры не меняются во время выполнения программы. Однако секции стека и кучи могут динамически сжиматься и увеличиваться во время выполнения программы. Каждый раз при вызове функции запись активации, содержащая параметры функции, локальные переменные и адрес возврата, помещается в стек; когда управление возвращается из функции, запись активации извлекается из стека. Аналогично, куча будет расти по мере динамического выделения памяти и уменьшаться, когда память возвращается в систему. Хотя секции стека и кучи растут навстречу друг другу, операционная система должна гарантировать, что они не перекрывают друг друга.

Подчеркнем, что программа сама по себе не является процессом. Программа — это пассивный объект, например файл, содержащий список инструкций, хранящихся на диске (часто

называемый исполняемым файлом). Напротив, процесс является активным объектом со счетчиком программ, указывающим следующую команду для выполнения и набор связанных ресурсов. Программа становится процессом, когда исполняемый файл загружается в память.

Два распространенных метода загрузки исполняемых файлов — это двойной щелчок по значку, представляющему исполняемый файл, и ввод имени исполняемого файла в командной строке (как в prog.exe или a.out). Хотя два процесса могут быть связаны с одной и той же программой, они, тем не менее, считаются двумя отдельными последовательностями выполнения. Например, несколько пользователей могут использовать разные копии почтовой программы или один и тот же пользователь может запускать множество копий программы веб-браузера. Каждый из них представляет собой отдельный процесс; и хотя текстовые разделы эквивалентны, разделы данных, кучи и стека различаются. Также часто бывает, что процесс порождает множество процессов во время своего выполнения. Мы обсуждаем такие вопросы в разделах 3.4.

Обратите внимание, что процесс сам по себе может быть средой выполнения для другого кода. Среда программирования Java представляет собой хороший пример. В большинстве случаев исполняемая программа Java выполняется внутри виртуальной машины Java (JVM). JVM выполняется как процесс, который интерпретирует загруженный Java-код и выполняет действия (посредством собственных машинных инструкций) от имени этого кода. Например, для запуска скомпилированной Java-программы Program.class, мы бы не стали использовать программу Java.

Команда java запускает JVM как обычный процесс, который, в свою очередь, выполняет программу Java на виртуальной машине. Концепция аналогична моделированию, за исключением того, что код пишется не для другого набора команд, а на языке Java.

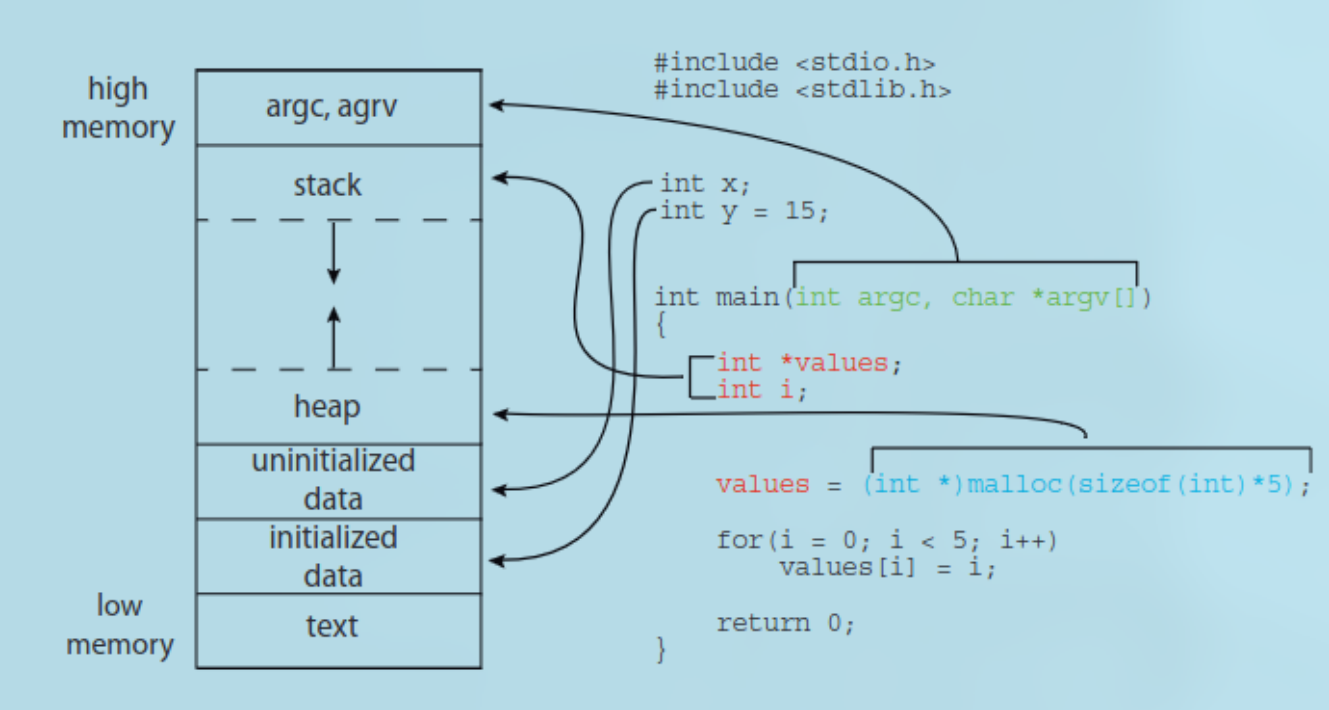
3.1.2 Состояние процесса

По мере выполнения процесса он меняет состояние. Состояние процесса частично определяется текущей деятельностью этого процесса. Процесс может находиться в одном из следующих состояний:

• Новый. Процесс создается.
• Запущенный. Указания выполняются.
• Ожидающий. Процесс ожидает возникновения какого-либо события (например, завершения ввода-вывода или приема сигнала).
• Готовый. Процесс ожидает назначения процессору.
• Прекращен. Процесс завершил выполнение.

РАСПОЛОЖЕНИЕ ПАМЯТИ С ПРОГРАММЫ

На рисунке ниже показано расположение программы на языке C в памяти, показывая, как различные разделы процесса связаны с реальной программой на языке C. Этот рисунок похож на общую концепцию процесса в памяти, показанную на рис. 3.1, с некоторыми отличиями:
• Раздел глобальных данных разделен на различные разделы для (a) инициализированных данных и (b) неинициализированных данных.
• Отдельный раздел выделен для параметров argc и argv, передаваемых в функцию main().



Команда GNU size может использоваться для определения размера (в байтах) некоторых из этих разделов. Предполагая, что имя исполняемого файла указанной выше программы C — это память, следующий результат генерируется при вводе команды «размер памяти»: текстовые данные bss dec шестнадцатеричное имя файла.

```
1158 284 8 1450 5aa memory
```

Поле данных относится к неинициализированным данным, а bss относится к инициализированным данным. (bss — это исторический термин, обозначающий блок, начинающийся с символа.)
Десятичные и шестнадцатеричные значения представляют собой сумму трех разделов, представленных в десятичном и шестнадцатеричном формате соответственно.

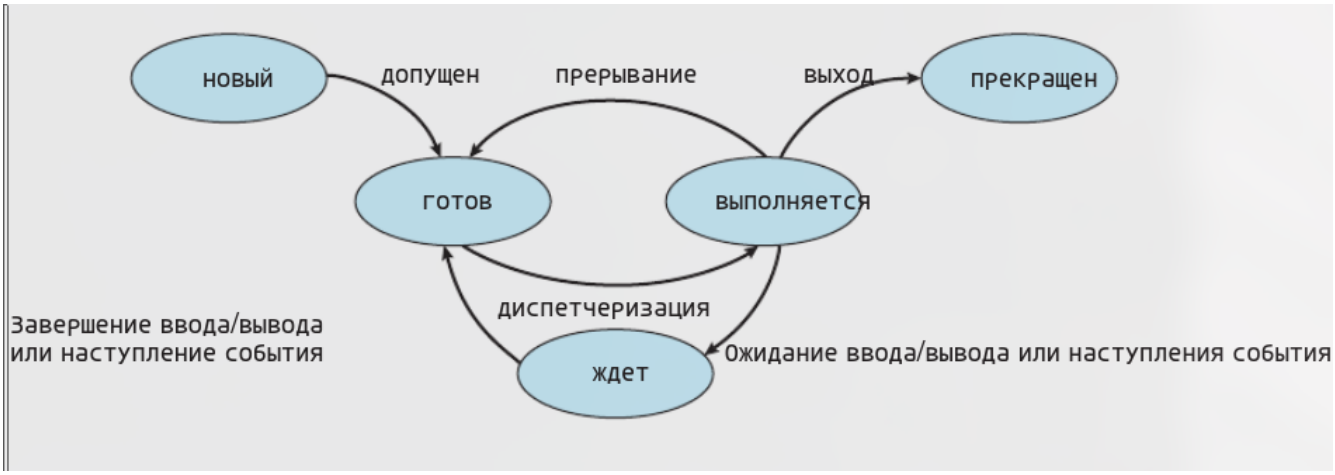


Рисунок 3.2 Диаграмма состояний процесса.

Эти имена произвольны и различаются в разных операционных системах.
Однако состояния, которые они представляют, встречаются во всех системах. Некоторые операционные системы также более точно определяют состояния процессов.
Важно понимать, что в любой момент времени на каждом ядре процессора может выполняться только один процесс. Однако многие процессы могут быть готовы и ждут.
Диаграмма состояний, соответствующая этим состояниям, представлена на рисунке 3.2.

3.1.3 Блок управления процессом

Каждый процесс представлен в операционной системе блоком управления процессом (PCB), также называемым блоком управления задачами. Печатная плата показана на рисунке 3.3.
Он содержит множество фрагментов информации, связанных с конкретным процессом, в том числе:
• Состояние процесса. Состояние может быть новым, готовым, запущенным, ожидающим, остановленным и т. д.
• Счетчик команд. Счетчик указывает адрес следующей инструкции, которая будет выполнена для этого процесса.

Рисунок 3.3 Блок управления процессом (PCB)

• Регистры ЦП. Регистры различаются по количеству и типу в зависимости от архитектуры компьютера. Они включают в себя аккумуляторы, индексные регистры, указатели стека и регистры общего назначения, а также любую информацию кода состояния.
Вместе со счетчиком программ эта информация о состоянии должна сохраняться при возникновении прерывания, чтобы обеспечить правильное продолжение процесса после его перепланирования запуска.

• Информация о планировании ЦП. Эта информация включает в себя приоритет процесса, указатели на очереди планирования и любые другие параметры планирования. (В главе 5 описывается планирование процессов.)
• Информация об управлении памятью. Эта информация может включать в себя такие элементы, как значения базового и предельного регистров, а также таблицы страниц или таблицы сегментов, в зависимости от системы памяти, используемой операционной системой (глава 9).
• Учетная информация. Эта информация включает в себя количество используемого процессора и реального времени, ограничения по времени, номера счетов, номера заданий или процессов и т. д.
• Информация о состоянии ввода-вывода. Эта информация включает в себя список устройств ввода-вывода, выделенных процессу, список открытых файлов и т. д. Короче говоря, печатная плата просто служит хранилищем всех данных, необходимых для запуска или перезапуска процесса, а также некоторых учетных данных.

3.1.4 Потоки

Обсуждавшаяся до сих пор модель процесса подразумевала, что процесс — это программа, выполняющая один поток выполнения. Например, когда процесс запускает программу текстового процессора, выполняется один поток инструкций. Этот единый поток управления позволяет процессу выполнять только одну задачу одновременно. Таким образом, пользователь не может одновременно вводить символы и запускать проверку орфографии. Большинство современных операционных систем расширили концепцию процесса, чтобы позволить процессу иметь несколько потоков выполнения и, таким образом, выполнять более одной задачи одновременно. Эта функция особенно полезна в многоядерных системах, где несколько потоков могут выполняться параллельно. Например, многопоточный текстовый процессор может назначить один поток для управления пользовательским вводом, в то время как другой поток будет выполнять проверку орфографии. В системах, поддерживающих потоки, PCB расширяется и включает информацию для каждого потока. Другие изменения во всей системе также необходимы для поддержки потоков. В главе 4 потоки подробно рассматриваются.

3.2 Планирование процессов

Цель мультипрограммирования состоит в том, чтобы какой-то процесс работал постоянно, чтобы максимизировать загрузку процессора. Целью разделения времени является настолько частое переключение ядра ЦП между процессами, чтобы пользователи могли взаимодействовать с каждой программой во время ее работы. Для достижения этих целей планировщик процессов выбирает доступный процесс (возможно, из набора нескольких доступных процессов) для выполнения программы на ядре. Каждое ядро ЦП может одновременно запускать один процесс.

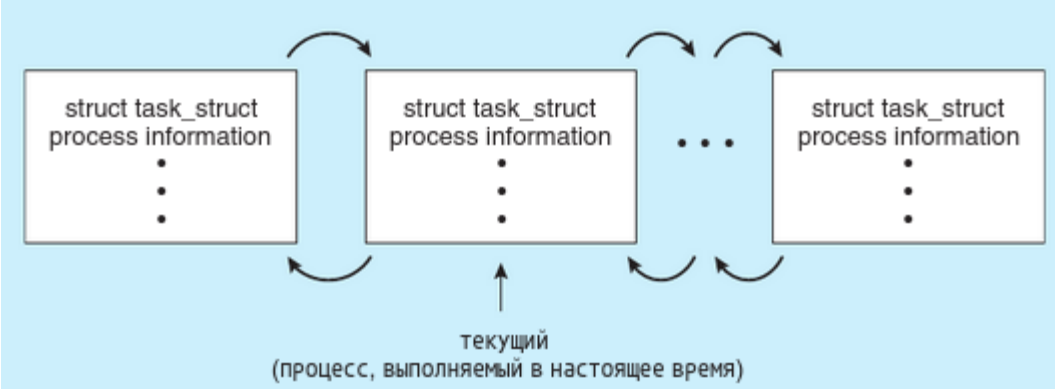
ПРЕДСТАВЛЕНИЕ ПРОЦЕССА В LINUX

Блок управления процессом в операционной системе Linux представлен структурой задачи C, которая находится в файле <include/linux/sched.h> включаемый файл в каталоге исходного кода ядра. Эта структура содержит всю необходимую информацию для представления процесса, включая состояние процесса, информацию о планировании и управлении памятью, список открытых файлов, а также указатели на родительский процесс и список его дочерних и однородных элементов. (Родителем процесса является процесс, который его создал; его

дочерними элементами являются любые процессы, которые он создает. Его братья и сестры являются дочерними элементами одного и того же родительского процесса.)
Некоторые из этих полей включают в себя:

```
long state; /* состояние процесса */
struct sched entity se; /* информация о расписании */
struct task struct *parent; /* родительский процесс */
struct list head children; /* дети этого процесса */
struct files struct *files; /* список открытых файлов */
struct mm struct *mm; /* адресное пространство */
```

Например, состояние процесса представлено в этой структуре длинным полем состояния.
В ядре Linux все активные процессы представлены в виде двусвязного списка структуры задач.
Ядро поддерживает указатель (текущий) на процесс, выполняющийся в данный момент в системе, как показано ниже:



В качестве иллюстрации того, как ядро может манипулировать одним из полей в структуре задачи для указанного процесса, предположим, что система хочет изменить состояние текущего процесса на значение new state.
Если current является указателем на выполняющийся в данный момент процесс, его состояние изменяется следующим образом:

```
current->state = new state;
```

В системе с одним ядром ЦП никогда не будет одновременно выполняться более одного процесса, тогда как в многоядерной системе одновременно может выполняться несколько процессов. Если процессов больше, чем ядер, лишним процессам придется ждать, пока ядро освободится и их можно будет перепланировать. Количество процессов, находящихся в данный момент в памяти, называется степенью мультипрограммирования.

Балансировка цели мультипрограммирования и разделения времени также требуют принятия во внимание общего поведения процесса. В общем, большинство процессов можно описать либо как связанные с вводом-выводом, либо как связанные с процессором.
Процесс, связанный с вводом-выводом, — это процесс, который тратит больше времени на ввод-вывод, чем на вычисления. Процесс, связанный с ЦП, напротив, генерирует запросы ввода-вывода нечасто, тратя больше времени на вычисления.

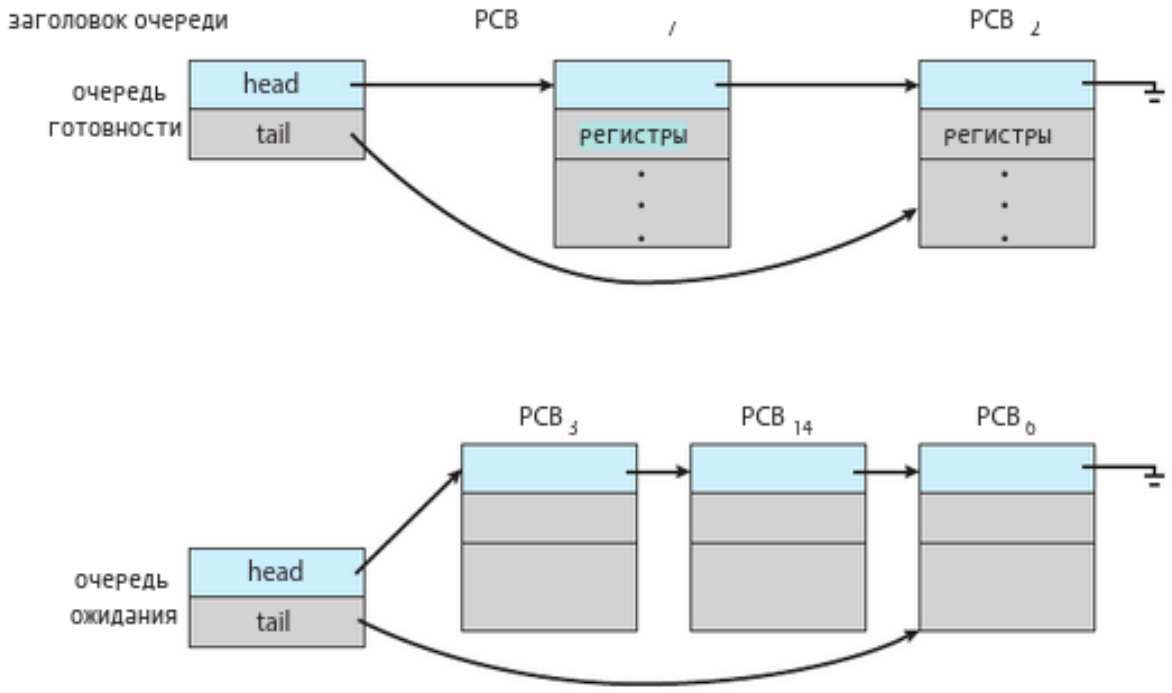


Рисунок 3.4 Очередь готовности и очерель ожидания

3.2.1 Планирование очередей

Когда процессы поступают в систему, они помещаются в очередь готовности, где они готовы и ожидают выполнения в ядре ЦП. Эта очередь обычно хранится в виде связанного списка; заголовок очереди готовности содержит указатели на первую PCB в списке, а каждая PCB включает поле указателя, указывающее на следующую плату в очереди готовности.

Система также включает в себя другие очереди. Когда процессу выделяется ядро ЦП, он выполняется некоторое время и в конечном итоге завершается, прерывается или ожидает возникновения определенного события, например завершения запроса ввода-вывода. Предположим, что процесс отправляет запрос ввода-вывода на такое устройство, как диск. Так как устройства работают значительно медленнее, чем процессоры, процессу придется ждать, пока ввод-вывод станет доступным. Процессы, ожидающие определенного события (например, завершения ввода-вывода), помещаются в очередь ожидания (рис. 3.4).

А распространенным представлением планирования процессов является диаграмма очередей, такая как показанная на рис. 3.5. Присутствуют два типа очередей: очередь готовности и набор очередей ожидания. Круги обозначают ресурсы, обслуживающие очереди, а стрелки указывают поток процессов в системе.

А новый процесс изначально помещается в очередь готовности. Он ждет там, пока не будет выбран для выполнения или отправлен. Как только процессу будет выделено ядро ЦП и он начнет выполняться, может произойти одно из нескольких событий:

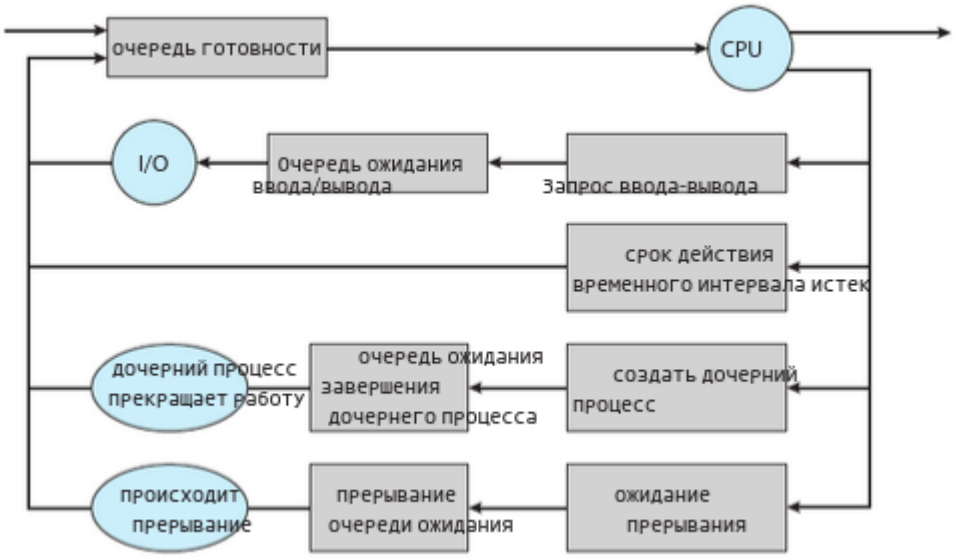


Рисунок 3.5 Диаграмма очередей при планировании процессов.

• Процесс может выдать запрос ввода-вывода, а затем быть помещен в очередь ожидания ввода-вывода.
• Процесс может создать новый дочерний процесс, а затем поместить его в очередь ожидания, пока он ожидает завершения дочернего процесса.
• Процесс может быть принудительно удален из ядра в результате прерывания или истечения его временного интервала и возвращен в очередь готовности.

В первых двух случаях процесс в конечном итоге переключается из состояния ожидания в состояние готовности, а затем снова помещается в очередь готовности. Процесс продолжает этот цикл до тех пор, пока не завершится, после чего он удаляется из всех очередей, а его плата и ресурсы освобождаются.

3.2.2 Планирование ЦП

На протяжении всего своего существования процесс перемещается между очередью готовности и различными очередями ожидания. Роль планировщика ЦП заключается в выборе процессов, находящихся в очереди готовности, и выделении ядра ЦП одному из них. Планировщик ЦП должен часто выбирать новый процесс для ЦП. Процесс, связанный с вводом-выводом, может выполняться всего несколько миллисекунд перед ожиданием запроса ввода-вывода. Хотя процесс, связанный с ЦП, потребует ядра ЦП в течение более длительного периода времени, планировщик вряд ли предоставит ядро процессу на длительный период. Вместо этого он, вероятно, предназначен для принудительного удаления ЦП из процесса и планирования запуска другого процесса. Таким образом, планировщик ЦП выполняется не реже одного раза в 100 миллисекунд, хотя обычно гораздо чаще.

Некоторые операционные системы имеют промежуточную форму планирования, известную как подкачка, основная идея которой заключается в том, что иногда может быть выгодно удалить процесс из памяти (и из активной борьбы за ЦП) и таким образом уменьшить степень мультипрограммирования. Позже процесс можно снова ввести в память и продолжить его выполнение с того места, где оно было остановлено. Эта схема известна как подкачка, поскольку процесс может быть «выгружен» из памяти на диск, где сохраняется его текущее состояние, а затем «выгружен» с диска обратно в память, где его состояние восстанавливается. Обмен обычно необходим только в том случае, если память перегружена и ее необходимо освободить. Обмен обсуждается в главе 9.

3.2.3 Переключение контекста

Как упоминалось в разделе 1.2.1, прерывания заставляют операционную систему переключать ядро ЦП с его текущей задачи и запускать процедуру ядра. Такие операции часто происходят в системах общего назначения. Когда происходит прерывание, системе необходимо сохранить текущий контекст процесса, выполняющегося в ядре ЦП, чтобы она могла восстановить этот контекст после завершения его обработки, по существу приостанавливая процесс, а затем возобновляя его. Контекст представлен на плате процесса. Он включает в себя значения регистров ЦП, состояние процесса (см. рис. 3.2) и информацию об управлении памятью. Как правило, мы выполняем сохранение текущего состояния ядра ЦП, будь то в режиме ядра или пользовательском режиме, а затем восстановление состояния для возобновления операций.

Переключение ядра ЦП на другой процесс требует сохранения состояния текущего процесса и восстановления состояния другого процесса. Эта задача известна как переключение контекста и проиллюстрирована на рисунке 3.6. Когда происходит переключение контекста, ядро сохраняет контекст старого процесса на своей плате и загружает сохраненный контекст нового процесса, запланированного к запуску. Время переключения контекста — это просто накладные расходы, поскольку при переключении система не выполняет никакой полезной работы. Скорость переключения варьируется от машины к машине, в зависимости от скорости памяти, количества регистров, которые необходимо скопировать, и наличия специальных инструкций (например, одной инструкции для загрузки или сохранения всех регистров). Типичная скорость составляет несколько микросекунд.

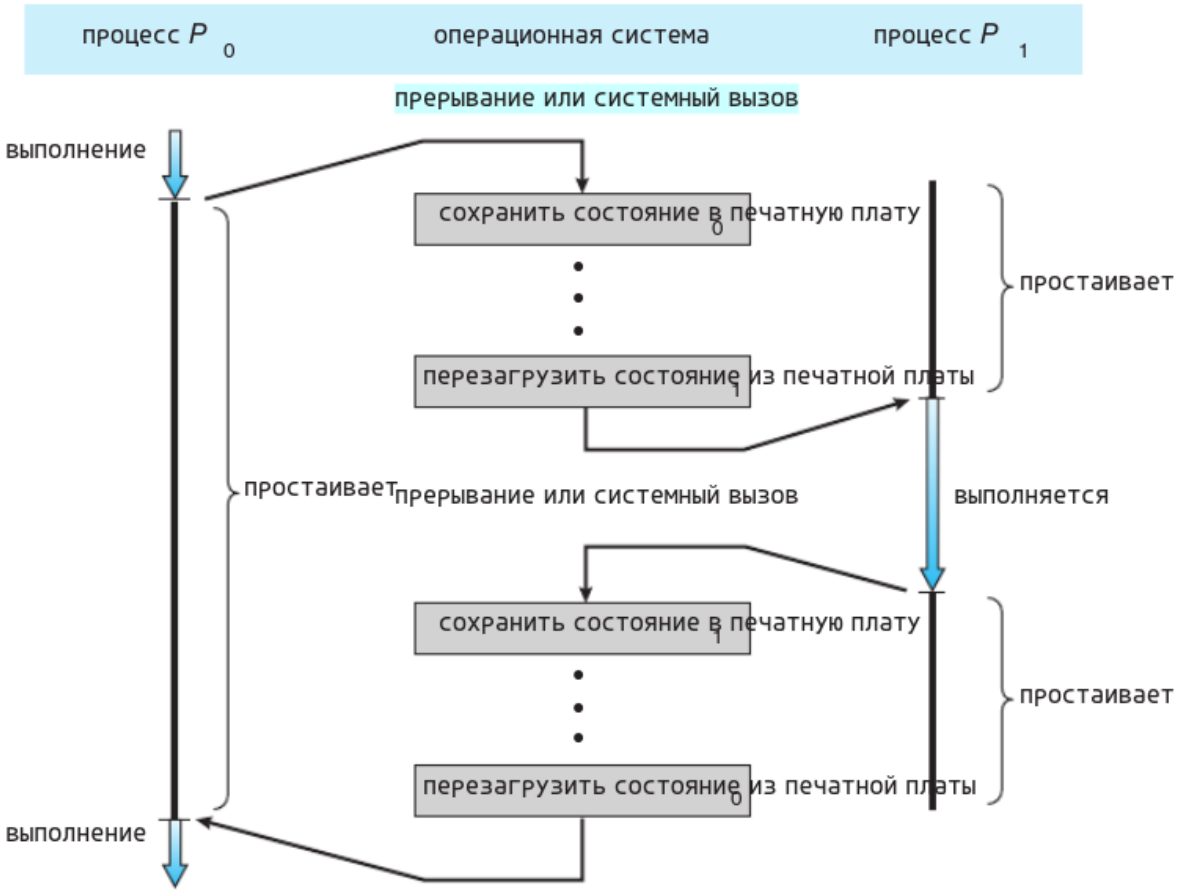


Рисунок 3.6 Диаграмма, показывающая переключение контекста с процесса на процесс

МНОГОЗАДАЧНОСТЬ В МОБИЛЬНЫХ СИСТЕМАХ

Из-за ограничений, налагаемых на мобильные устройства, ранние версии iOS не обеспечивали многозадачность пользовательских приложений; только одно приложение работало на переднем плане, а все остальные пользовательские приложения были приостановлены.

Задачи операционной системы были многозадачными, потому что они были написаны Apple и хорошо выполнялись. Однако, начиная с iOS 4, Apple предоставила ограниченную форму многозадачности для пользовательских приложений, что позволило одному приложению переднего плана работать одновременно с несколькими фоновыми приложениями.

(На мобильном устройстве приложением переднего плана является приложение, открытое в данный момент и отображаемое на дисплее. Фоновое приложение остается в памяти, но не занимает экран дисплея.) Программный API iOS 4 обеспечивал поддержку многозадачности, что позволяло процессу работать в фоновом режиме без приостановки. Однако он был ограничен и доступен только для нескольких типов приложений. Поскольку аппаратное обеспечение для мобильных устройств стало предлагать больший объем памяти, несколько вычислительных ядер и большее время автономной работы, последующие версии iOS начали поддерживать более широкие функциональные возможности для многозадачности с меньшими ограничениями. Например, больший экран планшетов iPad позволял одновременно запускать два приложения на переднем плане — метод, известный как разделенный экран.

С момента своего появления Android поддерживает многозадачность и не накладывает ограничений на типы приложений, которые могут работать в фоновом режиме.
Если приложению требуется обработка в фоновом режиме, оно должно использовать службу — отдельный компонент приложения, который запускается от имени фонового процесса.
Рассмотрим приложение потокового аудио: если приложение переходит в фоновый режим, служба продолжает отправлять аудиоданные драйверу аудиоустройства от имени фонового приложения. Фактически служба продолжит работать, даже если фоновое приложение приостановлено. Службы не имеют пользовательского интерфейса и занимают небольшой объем памяти, что обеспечивает эффективный метод многозадачности в мобильной среде.
Время переключения контекста сильно зависит от аппаратной поддержки. Например, некоторые процессоры предоставляют несколько наборов регистров. Переключение контекста здесь просто требует изменения указателя на текущий набор регистров. Конечно, если активных процессов больше, чем наборов регистров, система, как и раньше, прибегает к копированию данных регистров в память и из нее. Кроме того, чем сложнее операционная система, тем больший объем работы необходимо выполнить при переключении контекста. Как мы увидим в главе 9, передовые методы управления памятью могут потребовать переключения дополнительных данных с каждым контекстом. Например, адресное пространство текущего процесса должно быть сохранено, поскольку пространство следующей задачи готовится к использованию.
То, как сохраняется адресное пространство и какой объем работы необходимо для его сохранения, зависит от метода управления памятью операционной системы.

3.3 Операции над процессами

Процессы в большинстве систем могут выполняться одновременно, а также могут создаваться и удаляться динамически. Таким образом, эти системы должны обеспечивать механизм создания и завершения процессов. В этом разделе мы исследуем механизмы создания процессов и иллюстрируем создание процессов в системах UNIX и Windows.

3.3.1 Создание процесса

В ходе выполнения процесс может создать несколько новых процессов.
Как упоминалось ранее, создающий процесс называется родительским процессом, а новые процессы называются дочерними процессами этого процесса.
Каждый из этих новых процессов может, в свою очередь, создавать другие процессы, образуя дерево процессов.

Большинство операционных систем (включая UNIX, Linux и Windows) идентифицируют процессы по уникальному идентификатору процесса (или pid), который обычно представляет собой целое число. Pid предоставляет уникальное значение для каждого процесса в системе и может использоваться в качестве индекса для доступа к различным атрибутам процесса в ядре.

Рисунок 3.7 иллюстрирует типичное дерево процессов для операционной системы Linux, показывая имя каждого процесса и его идентификатор. (В этой ситуации мы довольно широко используем термин «процесс», поскольку Linux предпочитает вместо него термин «задача».)
Процесс systemd (pid которого всегда равен 1) служит корневым родительским процессом для всех пользовательских процессов и является первым пользовательским процессом, создаваемым при загрузке системы. После загрузки системы процесс systemd создает процессы, которые предоставляют дополнительные службы, такие как веб-сервер или сервер печати, ssh-сервер и т.п. На рисунке 3.7 мы видим двух дочерних элементов systemd — logind и sshd.
Процесс logind отвечает за управление клиентами, которые напрямую входят в систему.

В этом примере клиент вошел в систему и использует оболочку bash, которой присвоен идентификатор pid 8416. Используя интерфейс командной строки bash, этот пользователь создал процесс ps, а также редактор vim. Процесс sshd отвечает за управление клиентами, подключающимися к системе с помощью ssh (сокращение от Secure Shell).

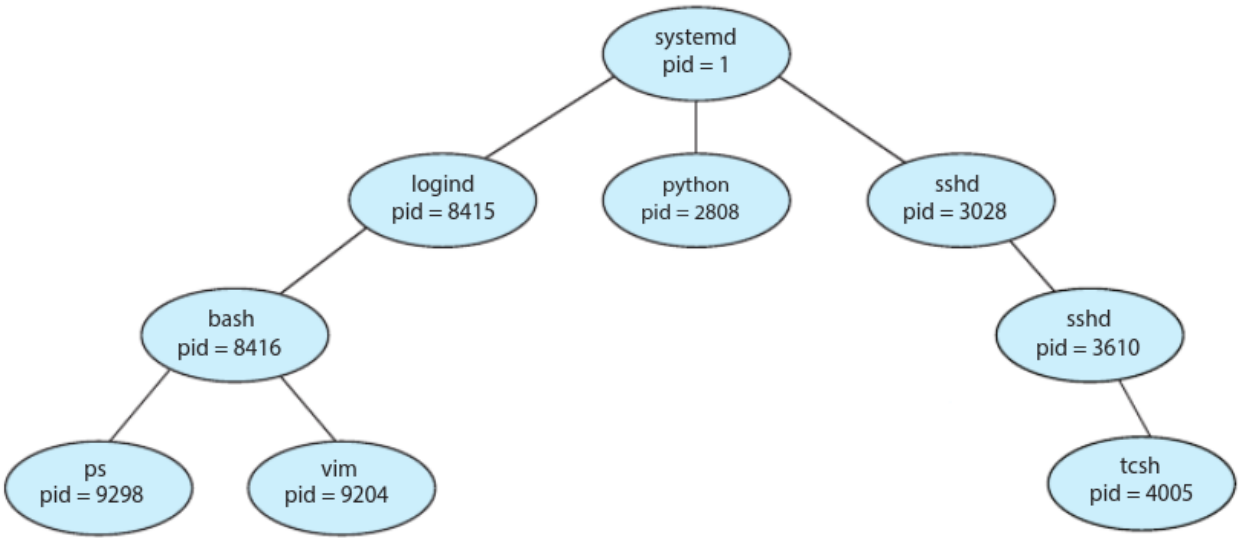


Рисунок 3.7 Дерево процессов в типичной системе Linux.

ПРОЦЕССЫ init И systemd
Традиционные системы UNIX идентифицируют процесс init как корневой для всех дочерних процессов. init (также известному как System V init) назначается pid, равный 1, и это первый процесс, создаваемый при загрузке системы. В дереве процессов, похожем на то, что показано на рис. 3.7, init находится в корне. Системы Linux изначально использовали подход инициализации System V, но в последних дистрибутивах он был заменен на systemd. Как описано в разделе 3.3.1, systemd служит начальным процессом системы, во многом аналогично System V init; однако он гораздо более гибок и может предоставлять больше услуг, чем init.

В системах UNIX и Linux мы можем получить список процессов с помощью команды ps. Например, команда

```
ps -el
```

отобразит полную информацию обо всех процессах, активных в данный момент в системе. А Дерево процессов, подобное показанному на рис. 3.7, может быть построено путем рекурсивного отслеживания родительских процессов вплоть до процесса systemd.
(Кроме того, в системах Linux предусмотрена команда pstree, которая отображает дерево всех процессов в системе.)

В общем, когда процесс создает дочерний процесс, этому дочернему процессу потребуются определенные ресурсы (процессорное время, память, файлы, устройства ввода-вывода) для выполнения своей задачи. Дочерний процесс может иметь возможность получать свои ресурсы непосредственно из операционной системы или может быть ограничен

подмножеством ресурсов родительского процесса. Родителю может потребоваться разделить свои ресурсы между своими дочерними элементами или он может иметь возможность совместно использовать некоторые ресурсы (например, память или файлы) между несколькими своими дочерними элементами. Ограничение дочернего процесса подмножеством родительских ресурсов предотвращает перегрузку системы любым процессом из-за создания слишком большого количества дочерних процессов. Помимо предоставления различных физических и логических ресурсов, родительский процесс может передавать данные инициализации (входные данные) дочернему процессу. Например, рассмотрим процесс, функция которого состоит в отображении содержимого файла, скажем, hw1.c, на экране терминала. При создании процесса он получит в качестве входных данных от родительского процесса имя файла hw1.c. Используя это имя файла, он откроет файл и запишет его содержимое. Он также может получить имя устройства вывода. Альтернативно, некоторые операционные системы передают ресурсы дочерним процессам. В такой системе новый процесс может получить два открытых файла, hw1.c и терминальное устройство, и может просто передать данные между ними.

Когда процесс создает новый процесс, существуют две возможности его выполнения:
1. Родитель продолжает выполняться одновременно со своими дочерними элементами.
2. Родитель ждет, пока некоторые или все его дочерние элементы завершатся.

Для нового процесса также есть две возможности адресного пространства:
1. Дочерний процесс является дубликатом родительского процесса (имеет ту же программу и данные, что и родительский).
2. В дочерний процесс загружена новая программа.

Чтобы проиллюстрировать эти различия, давайте сначала рассмотрим операционную систему UNIX.
В UNIX, как мы видели, каждый процесс идентифицируется своим идентификатором процесса, который представляет собой уникальное целое число.
Новый процесс создается системным вызовом fork().
Новый процесс состоит из копии адресного пространства исходного процесса.
Этот механизм позволяет родительскому процессу легко взаимодействовать со своим дочерним процессом.
Оба процесса (родительский и дочерний) продолжают выполнение инструкции после fork(), с одним отличием: код возврата fork() равен нулю для нового (дочернего) процесса, тогда как (ненулевой) идентификатор процесса ребенок возвращается родителю.
После системного вызова fork() один из двух процессов обычно использует системный вызов exec() для замены пространства памяти процесса новой программой.
Системный вызов exec() загружает двоичный файл в память (уничтожая образ памяти программы, содержащей системный вызов exec()) и начинает его выполнение.
Таким образом, два процесса могут взаимодействовать, а затем идти разными путями.
Родитель может затем создать больше детей; или, если ему больше нечего делать во время работы дочернего процесса, он может выполнить системный вызов wait(), чтобы выйти из очереди готовности до завершения дочернего процесса.
Поскольку вызов exec() перекрывает адресное пространство процесса новой программой, exec() не возвращает управление, пока не произойдет ошибка.

```
#include <sys/types.h>
#include <stdio.h>
```

```
#include <unistd.h>

int main()
{
pid t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { /* child process */
        execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
    }

    return 0;
}
```

Рисунок 3.8. Создание отдельного процесса с помощью системного вызова UNIX fork().

Программа на языке C, показанная на рисунке 3.8.
иллюстрирует ранее описанные системные вызовы UNIX.
Теперь у нас есть два разных процесса, выполняющих копии одной и той же программы. Единственное отличие состоит в том, что значение переменной pid для дочернего процесса равно нулю, а для родительского процесса — целое число, большее нуля (фактически это фактический pid дочернего процесса).
Дочерний процесс наследует привилегии и атрибуты планирования от родительского, а также определенные ресурсы, такие как открытые файлы.
Затем дочерний процесс перекрывает свое адресное пространство командой UNIX /bin/ls (используется для получения списка каталогов) с помощью системного вызова execlp() (execlp() — это версия системного вызова exec()). Родительский процесс ожидает завершения дочернего процесса с помощью системного вызова wait(). Когда дочерний процесс завершается (путем явного или неявного вызова функции выхода()), родительский процесс возобновляет работу с вызова метода wait(), где он завершается с использованием системного вызова exit(). Это также показано на рисунке 3.9. Конечно, ничто не мешает дочернему процессу не вызывать exec() и вместо этого продолжать выполняться как копия родительского процесса.
В этом сценарии родительский и дочерний процессы являются параллельными процессами, выполняющими одни и те же инструкции кода.
Поскольку дочерний процесс является копией родительского, каждый процесс имеет собственную копию любых данных. В качестве альтернативного примера мы рассмотрим создание процессов в Windows. Процессы создаются в Windows API с помощью функции CreateProcess(), которая аналогична функции fork() тем, что родительский процесс создает новый дочерний процесс. Однако,

тогда как fork() имеет дочерний процесс, наследующий адресное пространство своего родительского процесса, CreateProcess() требует загрузки указанной программы в адресное пространство дочернего процесса при создании процесса.
Более того, хотя fork() не передает никаких параметров, CreateProcess() ожидает не менее десяти параметров. Программа на языке C, показанная на рис. 3.10, иллюстрирует функцию CreateProcess(), которая создает дочерний процесс, загружающий приложение mspaint.exe.
Мы выбираем многие значения по умолчанию для десяти параметров, передаваемых в CreateProcess(). Читателям, заинтересованным в подробностях создания процессов и управления ими в Windows API, рекомендуется обратиться к библиографическим примечаниям в конце этой главы.
Два параметра, передаваемые в функцию CreateProcess(), являются экземплярами структур STARTUPINFO и PROCESS INFORMATION. STARTUPINFO определяет многие свойства нового процесса, такие как размер и внешний вид окна, а также обрабатывает стандартные файлы ввода и вывода. Структура PROCESS INFORMATION содержит дескриптор и идентификаторы вновь созданного процесса и его потока.
Мы вызываем функцию ZeroMemory(), чтобы выделить память для каждой из этих структур, прежде чем приступить к CreateProcess().

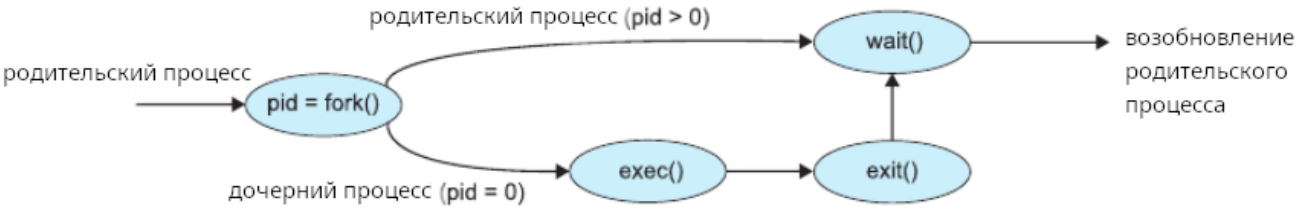


Рисунок 3.9 Создание процесса с помощью системного вызова fork()

```
#include <stdio.h>
#include <windows.h>
int main(VOID)
{
STARTUPINFO si;
PROCESS INFORMATION pi;

    /* выделение памяти */
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    /* создание дочернего процесса */
    if (!CreateProcess(NULL, /* используем командную строку */
        "C:\\WINDOWS\\system32\\mspaint.exe", /* команда */
        NULL, /* не наследуем обработку процесса */
        NULL, /* не наследуем управление потоком */
        FALSE, /* отключаем хэндл наследование */
        0, /* не создаем флаги */
        NULL, /* использем блок окружения родителя */
        NULL, /* используем родительскую существующую директорию */
        &si,
        &pi))
```

```
    {
        fprintf(stderr, "Create Process Failed");
        return -1;
    }
    /* родитель будет ожидать завершения процесса-ребенка */
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    /* закрываем хэндл */
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

Рисунок 3.10. Создание отдельного процесса с помощью Windows API.

Первые два параметра, передаваемые в CreateProcess(), — это имя приложения и параметры командной строки. Если имя приложения имеет значение NULL (как в этом случае), параметр командной строки указывает загружаемое приложение. В данном случае мы загружаем приложение Microsoft Windows mspaint.exe. Помимо этих двух начальных параметров, мы используем параметры по умолчанию для наследования дескрипторов процессов и потоков, а также для указания отсутствия флагов создания.
Мы также используем существующий блок среды родительского объекта и стартовый каталог. Наконец, мы предоставляем два указателя на структуры STARTUPINFO и PROCESS - INFORMATION, созданные в начале программы. На рис. 3.8 родительский процесс ожидает завершения дочернего процесса, вызывая системный вызов wait().
Эквивалентом этого в Windows является метод WaitForSingleObject(), которому передается дескриптор дочернего процесса — процесса pi.h — и который ожидает завершения этого процесса. После завершения дочернего процесса управление возвращается из функции WaitForSingleObject() в родительском процессе.

3.3.2 Завершение процесса

Процесс завершается, когда он завершает выполнение своего последнего оператора и просит операционную систему удалить его с помощью системного вызова exit(). В этот момент процесс может вернуть значение статуса (обычно целое число) своему ожидающему родительскому процессу (через системный вызов wait()). Все ресурсы процесса, включая физическую и виртуальную память, открытые файлы и буферы ввода-вывода, освобождаются и освобождаются операционной системой. Расторжение договора может произойти и при других обстоятельствах. Процесс может вызвать завершение другого процесса с помощью соответствующего системного вызова (например, TerminateProcess() в Windows).
Обычно такой системный вызов может быть вызван только родителем процесса, который должен быть завершен. В противном случае пользователь — или неправильно работающее приложение — может произвольно завершить процессы другого пользователя.
Обратите внимание, что родитель должен знать личности своих дочерних элементов, если он хочет их прекратить. Таким образом, когда один процесс создает новый процесс, идентификатор вновь созданного процесса передается родительскому процессу.

Родитель может прекратить выполнение одного из своих дочерних элементов по ряду причин, например:

• Дочерний элемент превысил использование некоторых ресурсов, которые ему были выделены. (Чтобы определить, произошло ли это, родительский элемент должен иметь механизм проверки состояния своих дочерних элементов.)
• Задача, поставленная перед ребенком, больше не требуется.
• Родительский процесс завершается, и операционная система не позволяет дочернему процессу продолжить работу, если его родительский элемент завершается.

Некоторые системы не позволяют дочернему элементу существовать, если его родительский элемент прекратил существование. В таких системах, если процесс завершается (нормально или ненормально), то все его дочерние процессы также должны быть завершены. Это явление, называемое каскадным завершением, обычно инициируется операционной системой.
Чтобы проиллюстрировать выполнение и завершение процесса, учтите, что в системах Linux и UNIX мы можем завершить процесс, используя системный вызов exit(), предоставляя статус завершения в качестве параметра:

```
/* выход со статусом 1 */
exit(1);
```

Фактически, при обычном завершении функция exit() будет вызываться либо напрямую (как показано выше), либо косвенно, поскольку библиотека времени выполнения C (которая добавляется к исполняемым файлам UNIX) будет включать вызов функции exit() по умолчанию.
Родительский процесс может дождаться завершения дочернего процесса, используя системный вызов wait(). Системному вызову wait() передается параметр, который позволяет родительскому элементу получить статус завершения дочернего процесса. Этот системный вызов также возвращает идентификатор завершенного дочернего процесса, чтобы родительский процесс мог определить, какой из его дочерних процессов завершился:

```
pid t pid;
int status;

pid = wait(&status);
```

Когда процесс завершается, его ресурсы освобождаются операционной системой.
Однако его запись в таблице процессов должна оставаться там до тех пор, пока родительский элемент не вызовет функцию wait(), поскольку таблица процессов содержит статус завершения процесса.
Процесс, который завершился, но родитель которого еще не вызвал функцию wait(), называется процессом-зомби. Все процессы переходят в это состояние, когда завершаются, но обычно они существуют как зомби лишь недолго. Как только родитель вызывает функцию wait(), идентификатор процесса-зомби и его запись в таблице процессов освобождаются.
Теперь представьте, что произойдет, если родительский процесс не вызовет функцию wait() и вместо этого завершится, тем самым оставив свои дочерние процессы сиротами.
Традиционные системы UNIX решают эту проблему, назначая процесс init новым родительским процессом для потерянных процессов. (Вспомните раздел 3.3.1, что init служит корнем иерархии процессов в системах UNIX.) Процесс init периодически вызывает wait(), тем самым позволяя собрать статус завершения любого потерянного процесса и освободить идентификатор потерянного процесса и запись в таблице процессов. Хотя в большинстве систем Linux init заменен на systemd, последний процесс по-прежнему может выполнять ту же роль, хотя Linux

также позволяет процессам, отличным от systemd, наследовать бесхозные процессы и управлять их завершением.

3.3.2.1 Иерархия процессов Android

Из-за ограничений ресурсов, таких как ограниченность памяти, мобильным операционным системам может потребоваться завершить существующие процессы, чтобы вернуть ограниченные системные ресурсы.
Вместо завершения произвольного процесса Android определил иерархию важности процессов, и когда системе необходимо завершить процесс, чтобы освободить ресурсы для нового или более важного процесса, она завершает процессы в порядке возрастания важности.
Иерархия классификаций процессов от наиболее к наименее важным выглядит следующим образом:
• Процесс переднего плана — текущий процесс, видимый на экране, представляющий приложение, с которым в данный момент взаимодействует пользователь.
• Видимый процесс — процесс, который не виден непосредственно на переднем плане, но который выполняет действие, на которое ссылается процесс переднего плана (т. е. процесс, выполняющий действие, статус которого отображается в процессе переднего плана).
• Служебный процесс — процесс, аналогичный фоновому процессу, но выполняющий действие, очевидное для пользователя (например, потоковая передача музыки). • Фоновый процесс — процесс, который может выполнять действие, но незаметное для пользователя.
• Пустой процесс — процесс, который не содержит активных компонентов, связанных с каким-либо приложением. Если необходимо освободить системные ресурсы, Android сначала завершит пустые процессы, затем фоновые процессы и т. д. Процессам присваивается рейтинг важности, и Android пытается присвоить процессу как можно более высокий рейтинг.
Например, если процесс предоставляет услугу и при этом является видимым, ему будет присвоен более важный видимый класс. Кроме того, практика разработки Android предполагает следование принципам жизненного цикла процесса. При соблюдении этих рекомендаций состояние процесса будет сохранено до его завершения и возобновлено в сохраненном состоянии, если пользователь вернется к приложению.

3.4 Межпроцессное взаимодействие

Процессы, выполняющиеся одновременно в операционной системе, могут быть как независимыми процессами, так и взаимодействующими процессами.
Процесс является независимым, если он не использует данные совместно с другими процессами, выполняющимися в системе. Процесс считается сотрудничающим, если он может влиять на другие процессы, выполняющиеся в системе, или подвергаться влиянию других процессов. Очевидно, что любой процесс, который обменивается данными с другими процессами, является взаимодействующим процессом.
Существует несколько причин для создания среды, позволяющей сотрудничать в процессах:
• Обмен информацией.
Поскольку несколько приложений могут быть заинтересованы в одной и той же информации (например, при копировании и вставке), мы должны предоставить среду, обеспечивающую одновременный доступ к такой информации.
• Ускорение вычислений.
Если мы хотим, чтобы конкретная задача выполнялась быстрее, мы должны разбить ее на подзадачи, каждая из которых будет выполняться параллельно с другими.

Обратите внимание, что такого ускорения можно добиться только в том случае, если компьютер имеет несколько вычислительных ядер.
• Модульность.
Возможно, мы захотим построить систему по модульному принципу, разделив системные функции на отдельные процессы или потоки, как мы обсуждали в главе 2.
Для взаимодействующих процессов требуется механизм межпроцессного взаимодействия (IPC), который позволит им обмениваться данными, то есть отправлять данные и получать данные друг от друга.

Существует две фундаментальные модели межпроцессного взаимодействия: общая память и передача сообщений. В модели общей памяти устанавливается область памяти, совместно используемая взаимодействующими процессами.
Затем процессы могут обмениваться информацией, считывая и записывая данные в общую область. В модели передачи сообщений общение происходит посредством сообщений, которыми обмениваются взаимодействующие процессы.
На рисунке 3.11 противопоставлены две модели коммуникации.

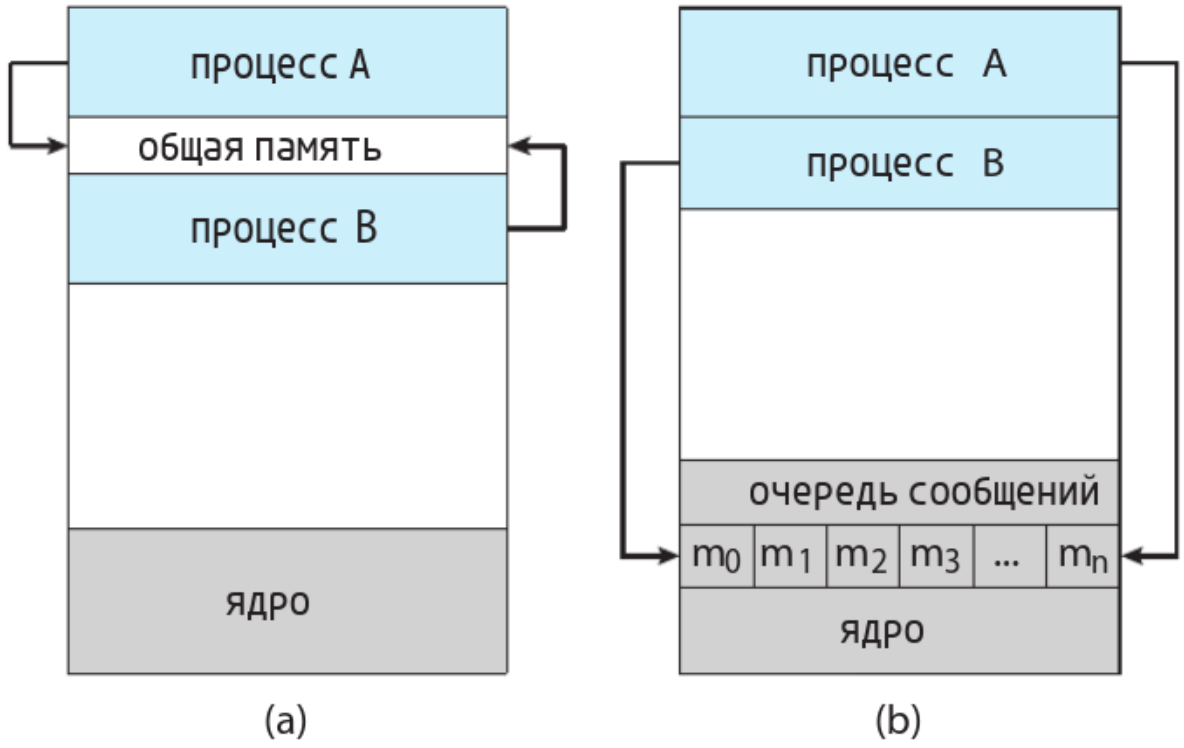


Рисунок 3.11 Модели коммуникаций. (a) Общая память. (b) Передача сообщений

МНОГОПРОЦЕССНАЯ АРХИТЕКТУРА — БРАУЗЕР Chrome
Многие веб-сайты содержат активный контент, такой как JavaScript, Flash и HTML5, что обеспечивает богатые и динамичные возможности просмотра веб-страниц.
К сожалению, эти веб-приложения также могут содержать программные ошибки, которые могут привести к замедлению времени отклика и даже к сбою веб-браузера.
Это не является большой проблемой для веб-браузера, который отображает контент только с одного веб-сайта.

Но большинство современных веб-браузеров поддерживают просмотр с вкладками, что позволяет одному экземпляру приложения веб-браузера открывать несколько веб-сайтов одновременно, при этом каждый сайт находится на отдельной вкладке. Для переключения между различными сайтами пользователю достаточно нажать на соответствующую вкладку. Эта схема проиллюстрирована ниже:

Проблема с этим подходом заключается в том, что если веб-приложение на любой вкладке дает сбой, то весь процесс, включая все другие вкладки, отображающие дополнительные веб-сайты, также дает сбой. Веб-браузер Google Chrome был разработан для решения этой проблемы за счет использования многопроцессной архитектуры. Chrome идентифицирует три различных типа процессов: браузер, средства рендеринга и плагины.

• Процесс браузера отвечает за управление пользовательским интерфейсом, а также за дисковый и сетевой ввод-вывод. Новый процесс браузера создается при запуске Chrome. Создается только один процесс браузера.
• Процессы рендеринга содержат логику для рендеринга веб-страниц. Таким образом, они содержат логику для обработки HTML, Javascript, изображений и т. д. Как правило, новый процесс рендеринга создается для каждого веб-сайта, открытого в новой вкладке, поэтому одновременно могут быть активны несколько процессов рендеринга.
• Процесс плагина создается для каждого типа используемого плагина (например, Flash или QuickTime). Процессы подключаемого модуля содержат код подключаемого модуля, а также дополнительный код, который позволяет подключаемому модулю взаимодействовать со связанными процессами рендеринга и процессом браузера.

Преимущество многопроцессного подхода заключается в том, что веб-сайты работают изолированно друг от друга. Если один веб-сайт выходит из строя, это затрагивает только процесс его рендеринга; все остальные процессы остаются нетронутыми. Более того, процессы рендеринга выполняются в «песочнице», а это означает, что доступ к диску и сетевому вводу-выводу ограничен, что сводит к минимуму последствия любых эксплойтов безопасности.
Обе только что упомянутые модели широко распространены в операционных системах, и многие системы реализуют обе. Передача сообщений полезна для обмена небольшими объемами данных, поскольку не нужно избегать конфликтов. Передачу сообщений также легче реализовать в распределенной системе, чем в общей памяти. (Хотя существуют системы, предоставляющие распределенную разделяемую память, мы не рассматриваем их в этом тексте.) Общая память может быть быстрее, чем передача сообщений, поскольку системы передачи сообщений обычно реализуются с использованием системных вызовов и, следовательно, требуют более трудоемкой задачи вмешательства ядра. В системах с общей памятью системные вызовы необходимы только для создания областей общей памяти. После того как общая память установлена, все обращения рассматриваются как обычные обращения к памяти, и никакая помощь со стороны ядра не требуется. В разделах 3.5 и 3.6 мы более подробно исследуем системы общей памяти и передачи сообщений.

3.5 IPC в системах с общей памятью

Межпроцессное взаимодействие с использованием общей памяти требует, чтобы взаимодействующие процессы установили область общей памяти. Обычно область общей памяти находится в адресном пространстве процесса, создающего сегмент общей памяти. Другие процессы, желающие взаимодействовать с использованием этого сегмента общей памяти, должны подключить его к своему адресному пространству. Напомним, что обычно операционная система пытается запретить одному процессу доступ к памяти другого процесса. Для общей памяти требуется, чтобы два или более процесса согласились снять это ограничение. Затем они могут обмениваться информацией, читая и записывая данные в общие области. Форма данных и их расположение определяются этими процессами и не контролируются операционной системой. Процессы также отвечают за то, чтобы они не записывали данные в одно и то же место одновременно.

Чтобы проиллюстрировать концепцию взаимодействующих процессов, давайте рассмотрим проблему производителя и потребителя, которая является общей парадигмой для взаимодействующих процессов. Процесс-производитель производит информацию, которая потребляется процессом-потребителем. Например, компилятор может создавать ассемблерный код, который используется ассемблером. Ассемблер, в свою очередь, может создавать объектные модули, которые используются загрузчиком.

Проблема производитель-потребитель также представляет собой полезную метафору парадигмы клиент-сервер. Обычно мы думаем о сервере как о производителе, а о клиенте как о потребителе. Например, веб-сервер создает (то есть предоставляет) веб-контент, такой как файлы HTML и изображения, которые потребляются (то есть читаются) клиентским веб-браузером, запрашивающим ресурс. Одним из решений проблемы производитель-потребитель является использование общей памяти. Чтобы позволить процессам производителя и потребителя работать одновременно, мы должны иметь доступный буфер элементов, который может быть заполнен производителем и очищен потребителем. Этот буфер будет находиться в области памяти, которая используется совместно процессами-производителями и потребителями. Производитель может производить один товар, в то время как потребитель потребляет другой товар. Производитель и потребитель должны быть синхронизированы, чтобы потребитель не пытался потреблять еще не произведенный товар.

Можно использовать два типа буферов. Неограниченный буфер не накладывает практических ограничений на размер буфера. Потребителю, возможно, придется ждать новых товаров, но производитель всегда может произвести новые товары. Ограниченный буфер предполагает фиксированный размер буфера. В этом случае потребитель должен ждать, если буфер пуст, а производитель должен ждать, если буфер полон.

Давайте более подробно рассмотрим, как ограниченный буфер иллюстрирует межпроцессное взаимодействие с использованием общей памяти. Следующие переменные находятся в области памяти, совместно используемой процессами-производителями и потребителями:

```
#define BUFFER SIZE 10
typedef struct {
. . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

Общий буфер реализован как кольцевой массив с двумя логическими указателями: входным и выходным. Переменная in указывает на следующую свободную позицию в буфере; out указывает на первую полную позицию в буфере. Буфер пуст, когда in == out; буфер полон, когда ((in + 1) % BUFFER_SIZE) == out. Код процесса-производителя показан на рисунке 3.12, а код процесса-потребителя показан на рисунке 3.13. Далее у процесса-производителя создается локальная переменная, в которой сохраняется новый создаваемый элемент. Процесс-потребитель имеет следующую потребляемую локальную переменную, в которой хранится потребляемый элемент. Эта схема допускает одновременное размещение в буфере не более 1 элемента в размере BUFFER_SIZE. Мы оставляем вам это в качестве упражнения, чтобы найти решение, в котором элементы BUFFER_SIZE могут одновременно находиться в буфере.
В разделе 3.7.1 мы иллюстрируем POSIX API для разделяемой памяти.

```
item next produced;
while (true) {
    /* произвести элемент в следующем произведении */
    while (((in + 1) % BUFFER SIZE) == out)
        ; /* ничего не делать */
    buffer[in] = next produced;
    in = (in + 1) % BUFFER SIZE;
}
```

Рисунок 3.12. Процесс-производитель, использующий общую память.

Одна из проблем, которую не решает эта иллюстрация, касается ситуации, в которой и процесс-производитель, и процесс-потребитель пытаются одновременно получить доступ к общему буферу.
В главах 6 и 7 мы обсуждаем, как можно эффективно реализовать синхронизацию взаимодействующих процессов в среде с общей памятью.

3.6 IPC в системах передачи сообщений

В разделе 3.5 мы показали, как взаимодействующие процессы могут взаимодействовать в среде с общей памятью. Схема требует, чтобы эти процессы совместно использовали область памяти и чтобы код для доступа к общей памяти и управления ею был явно написан программистом приложения. Другой способ добиться того же эффекта — предоставить операционной системе средства взаимодействия процессов для взаимодействия друг с другом через средство передачи сообщений.

```
item next consumed;

while (true) {
    while (in == out)
        ; /* ничего не делать */

    next consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* потребляем предмет при следующем потреблении */
}
```

Рисунок 3.13. Процесс-потребитель, использующий общую память.

Передача сообщений предоставляет механизм, позволяющий процессам взаимодействовать и синхронизировать свои действия, не используя одно и то же адресное пространство. Это особенно полезно в распределенной среде, где взаимодействующие процессы могут находиться на разных компьютерах, подключенных к сети. Например, программа интернет-чата может быть разработана таким образом, чтобы участники чата общались друг с другом путем обмена сообщениями. Средство передачи сообщений обеспечивает как минимум две операции:

```
send(message)
```
и
```
receive(message)
```

Сообщения, отправляемые процессом, могут иметь фиксированный или переменный размер. Если можно отправлять только сообщения фиксированного размера, реализация на системном уровне проста. Однако это ограничение усложняет задачу программирования. И наоборот, сообщения переменного размера требуют более сложной реализации на системном уровне, но задача программирования становится проще. Это обычный компромисс, наблюдаемый при проектировании операционных систем. Если процессы P и Q хотят взаимодействовать, они должны отправлять сообщения и получать сообщения друг от друга: между ними должна существовать связь. Эту ссылку можно реализовать разными способами.
Здесь нас интересует не физическая реализация канала (например, общая память, аппаратная шина или сеть, которые рассматриваются в главе 19), а скорее его логическая реализация.
Вот несколько методов логической реализации ссылки и операций send()/receive():

• Прямое или косвенное общение
• Синхронная или асинхронная связь
• Автоматическая или явная буферизация.

Далее мы рассмотрим проблемы, связанные с каждой из этих функций.

3.6.1 Наименование

Процессы, которые хотят взаимодействовать, должны иметь возможность ссылаться друг на друга. Они могут использовать как прямое, так и косвенное общение.
При прямом общении каждый процесс, желающий взаимодействовать, должен явно указать получателя или отправителя сообщения. В этой схеме примитивы send() и получения() определяются как:

• send(P, message) — отправить сообщение для обработки P.
• get(Q, message) — получить сообщение от процесса Q.

Канал связи в этой схеме имеет следующие свойства:

• Связь устанавливается автоматически между каждой парой процессов, желающих взаимодействовать.
Для связи процессам необходимо знать только личность друг друга.
• Ссылка связана ровно с двумя процессами.

• Между каждой парой процессов существует ровно одно соединение.

Эта схема демонстрирует симметрию адресации; то есть оба – процесс-отправитель и процесс-получатель должен назвать друг друга для связи. Вариант этой схемы использует асимметрию адресации. Здесь только отправитель называет получателя; получатель не обязан называть отправителя. В этой схеме примитивы send() и receive() определяются следующим образом:

• send(P, message) — отправить сообщение для обработки P.
• receive(id, message) — получить сообщение от любого процесса. Идентификатор переменной установлен на имя процесса, с которым установлена связь.

Недостатком обеих этих схем (симметричной и асимметричной) является ограниченная модульность результирующих определений процессов. Изменение идентификатора процесса может потребовать изучения всех других определений процесса. Все ссылки на старый идентификатор должны быть найдены, чтобы их можно было изменить на новый идентификатор. В общем, любые подобные методы жесткого кодирования, в которых идентификаторы должны быть указаны явно, менее желательны, чем методы, включающие косвенность, как описано ниже.

При непрямой связи сообщения отправляются и принимаются из почтовых ящиков или портов. Почтовый ящик можно рассматривать абстрактно как объект, в который процессы могут помещать сообщения и из которого сообщения могут быть удалены. Каждый почтовый ящик имеет уникальный идентификатор. Например, очереди сообщений POSIX используют целочисленное значение для идентификации почтового ящика. Процесс может взаимодействовать с другим процессом через несколько разных почтовых ящиков, но два процесса могут взаимодействовать только в том случае, если у них есть общий почтовый ящик. Примитивы send() и get() определяются следующим образом:

• send(A, message) — отправить сообщение в почтовый ящик A.
• get(A, message) — получить сообщение из почтового ящика A.

В этой схеме канал связи имеет следующие свойства:

• Связь между парой процессов устанавливается только в том случае, если оба члена пары имеют общий почтовый ящик.
• Ссылка может быть связана с более чем двумя процессами.
• Между каждой парой взаимодействующих процессов может существовать несколько различных ссылок, каждая из которых соответствует одному почтовому ящику.

Теперь предположим, что процессы P1, P2 и P3 используют общий почтовый ящик A.
Процесс P1 отправляет сообщение A, в то время как P2 и P3 выполняют метод получения() от A.
Какой процесс получит сообщение, отправленное P1?
Ответ зависит от того, какой из следующих методов мы выберем:

• Разрешить связывание ссылки максимум с двумя процессами.
• Разрешить не более одному процессу одновременно выполнять операцию получения().
• Разрешить системе произвольно выбирать, какой процесс получит сообщение (то есть сообщение получит либо P2, либо P3, но не оба).

Система может определить алгоритм выбора процесса, который получит сообщение (например, циклический перебор, когда процессы по очереди получают сообщения).
Система может идентифицировать получателя для отправителя.

Почтовый ящик может принадлежать либо процессу, либо операционной системе. Если почтовый ящик принадлежит процессу (то есть ящик является частью адресного пространства процесса), то мы различаем владельца (который может получать сообщения только через этот ящик) и пользователя (который может только отправлять сообщения в почтовый ящик). Поскольку у каждого почтового ящика есть уникальный владелец, не может быть путаницы в том, какой процесс должен получить сообщение, отправленное в этот почтовый ящик.
Когда процесс, владеющий почтовым ящиком, завершается, почтовый ящик исчезает.
Любой процесс, который впоследствии отправляет сообщение в этот почтовый ящик, должен быть уведомлен о том, что почтовый ящик больше не существует.

Напротив, почтовый ящик, принадлежащий операционной системе, существует самостоятельно. Он независим и не привязан к какому-либо конкретному процессу.
Затем операционная система должна предоставить механизм, который позволяет процессу делать следующее:

• Создайте новый почтовый ящик.
• Отправлять и получать сообщения через почтовый ящик.
• Удалить почтовый ящик.

Процесс создания нового почтового ящика по умолчанию является владельцем этого почтового ящика. Изначально владелец — единственный процесс, который может получать сообщения через этот почтовый ящик. Однако права владения и получения могут быть переданы другим процессам посредством соответствующих системных вызовов. Конечно, это положение может привести к использованию нескольких получателей для каждого почтового ящика.

3.6.2 Синхронизация

Связь между процессами осуществляется посредством вызовов примитивов send() и получения(). Существуют разные варианты реализации каждого примитива.
Передача сообщений может быть как блокирующей, так и неблокирующей, также известной как синхронная и асинхронная. (В этом тексте вы встретите понятия синхронного и асинхронного поведения по отношению к различным алгоритмам операционной системы.)

• Блокировка отправки. Процесс отправки блокируется до тех пор, пока сообщение не будет получено принимающим процессом или почтовым ящиком.
• Неблокирующая отправка. Процесс отправки отправляет сообщение и возобновляет работу.
• Блокировка приема. Получатель блокируется до тех пор, пока сообщение не станет доступным.
• Неблокирующий прием. Получатель получает либо допустимое сообщение, либо нулевое.

```
message next produced;

while (true) {
    /* произвести элемент в следующем произведении */

    send(next produced);
}
```

Рисунок 3.14. Процесс производителя, использующий передачу сообщений.

Возможны различные комбинации send() и получения(). Когда и send(), и get() блокируются, между отправителем и получателем происходит рандеву. Решение проблемы производитель-потребитель становится тривиальным, если мы используем блокирующие операторы send() и get(). Производитель просто вызывает блокирующий вызов send() и ждет, пока сообщение не будет доставлено получателю или почтовому ящику. Аналогично, когда потребитель вызывает метод get(), он блокируется до тех пор, пока сообщение не станет доступным. Это показано на рисунках 3.14 и 3.15.

3.6.3 Буферизация

Независимо от того, является ли связь прямой или косвенной, сообщения, которыми обмениваются взаимодействующие процессы, находятся во временной очереди.
В принципе, такие очереди можно реализовать тремя способами:

• Нулевая мощность. Очередь имеет максимальную длину, равную нулю; таким образом, в ссылке не может быть ожидающих сообщений. В этом случае отправитель должен заблокировать сообщение до тех пор, пока получатель не получит сообщение.
• Ограниченная мощность. Очередь имеет конечную длину n; таким образом, в нем может находиться не более n сообщений. Если очередь не заполнена при отправке нового сообщения, сообщение помещается в очередь (либо сообщение копируется, либо указатель на сообщение сохраняется), и отправитель может продолжить выполнение без ожидания.
Однако пропускная способность канала ограничена. Если ссылка заполнена, отправитель должен заблокировать ее до тех пор, пока в очереди не освободится место.
• Неограниченная мощность. Длина очереди потенциально бесконечна; таким образом, в нем может ожидать любое количество сообщений. Отправитель никогда не блокируется.
Случай с нулевой пропускной способностью иногда называют системой сообщений без буферизации. Остальные случаи называются системами с автоматической буферизацией.

Случай с нулевой пропускной способностью иногда называют системой сообщений без буферизации. Остальные случаи называются системами с автоматической буферизацией.

```
message next consumed;

while (true) {
    receive(next consumed);
    /* потребляем предмет при следующем потреблении */
}
```

Рисунок 3.15. Процесс-потребитель, использующий передачу сообщений.

3.7 Примеры систем IPC

В этом разделе мы рассмотрим четыре различные системы IPC.

Сначала мы рассмотрим POSIX API для общей памяти, а затем обсудим передачу сообщений в операционной системе Mach. Далее мы представляем Windows IPC, которая интересно использует общую память как механизм обеспечения определенных типов передачи сообщений. В заключение мы рассмотрим каналы — один из самых ранних механизмов IPC в системах UNIX.

3.7.1 Общая память POSIX

Для систем POSIX доступно несколько механизмов IPC, включая разделяемую память и передачу сообщений.
Здесь мы исследуем POSIX API для общей памяти.
Общая память POSIX организована с использованием отображенных в память файлов, которые связывают область общей памяти с файлом.
Сначала процесс должен создать объект общей памяти с помощью системного вызова shm open() следующим образом:

```
fd = shm open(name, O CREAT | O RDWR, 0666);
```

Первый параметр указывает имя объекта общей памяти.
Процессы, желающие получить доступ к этой общей памяти, должны обращаться к объекту по этому имени. Последующие параметры указывают, что объект общей памяти должен быть создан, если он еще не существует (O CREAT), и что объект открыт для чтения и записи (O RDWR). Последний параметр устанавливает права доступа к файлу объекта общей памяти. Успешный вызов shm open() возвращает целочисленный файловый дескриптор объекта общей памяти. После создания объекта функция ftruncate() используется для настройки размера объекта в байтах. Вызов

```
ftruncate(fd, 4096);
```

устанавливает размер объекта в 4096 байт.

Наконец, функция mmap() создает отображенный в памяти файл, содержащий объект общей памяти. Он также возвращает указатель на файл, отображенный в памяти, который используется для доступа к объекту общей памяти.

Программы, показанные на рисунках 3.16 и 3.17, используют модель производитель-потребитель при реализации общей памяти.
Производитель создает объект с общей памятью и записывает в общую память, а потребитель читает из общей памяти.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

#include <sys/mman.h>

int main()
{
/* размер (в байтах) объекта общей памяти */
```

```
const int SIZE = 4096;
/* имя объекта общей памяти */
const char *name = "OS";
/* строки, записываемые в общую память */
const char *message 0 = "Hello";
const char *message 1 = "World!";

/* дескриптор файла общей памяти */
int fd;
/* указатель на объект общей памяти */
char *ptr;

    /* создаем объект общей памяти */
    fd = shm open(name,O CREAT | O RDWR,0666);

    /* настраиваем размер объекта общей памяти */
    ftruncate(fd, SIZE);

    /* сопоставляем память с объектом общей памяти */
    ptr = (char *)
        mmap(0, SIZE, PROT READ | PROT WRITE, MAP SHARED, fd, 0);

    /* запись в объект общей памяти */
    sprintf(ptr,"%s",message 0);
    ptr += strlen(message 0);
    sprintf(ptr,"%s",message 1);
    ptr += strlen(message 1);

    return 0;
}
```

Рисунок 3.16. Процесс производителя, иллюстрирующий API общей памяти POSIX.

Производитель, показанный на рис. 3.16, создает объект общей памяти с именем OS и записывает печально известную строку «Hello World!» в общую память.
Программа отображает в памяти объект общей памяти указанного размера и позволяет производить запись в этот объект. Флаг MAP SHARED указывает, что изменения объекта общей памяти будут видны всем процессам, совместно использующим этот объект.
Обратите внимание, что мы записываем в объект общей памяти, вызывая функцию sprintf() и записывая форматированную строку в указатель ptr. После каждой записи мы должны увеличивать указатель на количество записанных байт.

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

#include <sys/mman.h>

int main()
{
```

```
/* размер (в байтах) объекта общей памяти */
const int SIZE = 4096;
/* имя объекта общей памяти */
const char *name = "OS";
/* дескриптор файла общей памяти */
int fd;
/* указатель на объект общей памяти */
char *ptr;

    /* открываем объект общей памяти */
    fd = shm open(name, O RDONLY, 0666);

    /* сопоставляем память с объектом общей памяти */
    ptr = (char *)
        mmap(0, SIZE, PROT READ | PROT WRITE, MAP SHARED, fd, 0);

    /* чтение из объекта общей памяти */
    printf("%s",(char *)ptr);

    /* удаляем объект общей памяти */
    shm unlink(name);

    return 0;
}
```

Рисунок 3.17. Потребительский процесс, иллюстрирующий API общей памяти POSIX.

Процесс-потребитель, показанный на рис. 3.17, считывает и выводит содержимое разделяемой памяти. Потребитель также вызывает функцию shm unlink(), которая удаляет сегмент общей памяти после того, как потребитель получил к нему доступ. Дополнительные упражнения с использованием API общей памяти POSIX мы предоставим в упражнениях по программированию в конце этой главы. Кроме того, мы более подробно рассмотрим отображение памяти в разделе 13.5.

3.7.2 Передача сообщения Mach

В качестве примера передачи сообщений мы далее рассмотрим операционную систему Mach. Mach был специально разработан для распределенных систем, но оказался пригодным также для настольных и мобильных систем, о чем свидетельствует его включение в операционные системы Mac OS и iOS, как обсуждалось в главе 2.

Ядро Mach поддерживает создание и уничтожение множества задач, которые похожи на процессы, но имеют несколько потоков управления и меньше связанных ресурсов. Большая часть коммуникации в Mach, включая всю межзадачную связь, осуществляется посредством сообщений. Сообщения отправляются и принимаются из почтовых ящиков, которые в Mach называются портами. Порты имеют конечный размер и являются однонаправленными; при двусторонней связи сообщение отправляется на один порт, а ответ отправляется на отдельный порт ответа. Каждый порт может иметь несколько отправителей, но только один получатель. Mach использует порты для представления таких ресурсов, как задачи, потоки, память и процессоры, а передача сообщений обеспечивает объектно-ориентированный подход для взаимодействия с этими системными ресурсами и службами. Передача сообщений может

происходить между любыми двумя портами на одном и том же хосте или на разных хостах в распределенной системе.

С каждым портом связан набор прав порта, которые определяют возможности, необходимые задаче для взаимодействия с портом. Например, чтобы задача могла получить сообщение из порта, она должна иметь возможность MACH_PORT_RIGHT_RECEIVE для этого порта. Задача, создающая порт, является владельцем этого порта, а владелец — единственная задача, которой разрешено получать сообщения с этого порта. Владелец порта также может манипулировать возможностями порта. Чаще всего это делается при установке ответного порта. Например, предположим, что задача T1 владеет портом P1 и отправляет сообщение на порт P2, который принадлежит задаче T2. Если T1 ожидает получить ответ от T2, он должен предоставить T2 правильный MACH PORT RIGHT SEND для порта P1. Владение правами порта осуществляется на уровне задачи, что означает, что все потоки, принадлежащие одной задаче, имеют одни и те же права порта. Таким образом, два потока, принадлежащие одной и той же задаче, могут легко взаимодействовать путем обмена сообщениями через порт каждого потока, связанный с каждым потоком. При создании задачи также создаются два специальных порта — порт Task Self и порт Notify. Ядро имеет права на получение порта Task Self, который позволяет задаче отправлять сообщения ядру. Ядро может отправлять уведомления о возникновении событий на порт Notify задачи (на который, конечно же, задача имеет права получения). Вызов функции mach port allocate() создает новый порт и выделяет место для очереди сообщений. Он также определяет права на порт. Каждое право порта представляет имя этого порта, и доступ к порту возможен только через право. Имена портов представляют собой простые целочисленные значения и ведут себя так же, как файловые дескрипторы UNIX. Следующий пример иллюстрирует создание порта с использованием этого API:

```
mach_port_t port; // имя порта справа

mach_port_allocate(
    mach_task_self(), // задача, ссылающаяся на себя
    MACH_PORT_RIGHT_RECEIVE, // права для этого порта
    &port); // имя нужного порта
```

Каждая задача также имеет доступ к порту начальной загрузки, что позволяет задаче зарегистрировать созданный ею порт на общесистемном сервере начальной загрузки.
После регистрации порта на загрузочном сервере другие задачи смогут искать порт в этом реестре и получать права на отправку сообщений на этот порт.

Очередь, связанная с каждым портом, имеет конечный размер и изначально пуста.
Когда сообщения отправляются в порт, они копируются в очередь.
Все сообщения доставляются надежно и имеют одинаковый приоритет.
Mach гарантирует, что несколько сообщений от одного и того же отправителя будут поставлены в очередь в порядке «первым поступило — первым обслужено» (FIFO), но не гарантирует абсолютный порядок. Например, сообщения от двух отправителей могут быть поставлены в очередь в любом порядке.

Сообщения Mach содержат следующие два поля:

• Заголовок сообщения фиксированного размера, содержащий метаданные о сообщении, включая размер сообщения, а также порты источника и назначения.
Обычно отправляющий поток ожидает ответа, поэтому имя порта источника передается принимающей задаче, которая может использовать его в качестве «обратного адреса» при отправке ответа.

• Тело переменного размера, содержащее данные.

Сообщения могут быть простыми или сложными. Простое сообщение содержит обычные неструктурированные пользовательские данные, которые не интерпретируются ядром. Сложное сообщение может содержать указатели на ячейки памяти, содержащие данные (так называемые «внеочередные» данные), или также может использоваться для передачи прав порта другой задаче. Указатели вне строки данных особенно полезны, когда сообщение должно передавать большие фрагменты данных. Простое сообщение потребует копирования и упаковки данных в сообщении; Для передачи данных вне линии требуется только указатель, который ссылается на ячейку памяти, в которой хранятся данные.

Функция mach msg() — это стандартный API для отправки и получения сообщений. Значение одного из параметров функции — MACH SEND MSG или MACH RCV MSG — указывает, является ли это операцией отправки или получения. Теперь мы покажем, как он используется, когда клиентская задача отправляет простое сообщение серверной задаче. Предположим, что есть два порта — клиентский и серверный — связанные с задачами клиента и сервера соответственно. Код на рисунке 3.18 показывает клиентскую задачу, создающую заголовок и отправляющую сообщение на сервер, а также серверную задачу, получающую сообщение, отправленное от клиента.

Вызов функции mach msg() вызывается пользовательскими программами для передачи сообщений. Затем mach msg() вызывает функцию mach msg capture(), которая является системным вызовом ядра Mach. В ядре mach msg capture() затем вызывает функцию mach msg overwriterap(), которая затем обрабатывает фактическую передачу сообщения.

```
#include <mach/mach.h>

struct message {
    mach_msg_header_t header; // Заголовок сообщения
    int data;
};

mach_port_t client; // Клиентский порт
mach_port_t server; // Серверный порт

/* Код клиента */

struct message message;

// Составление заголовка
message.header.msgh_size = sizeof(message); // размер сообщения
message.header.msgh_remote_port = server; // удаленный порт
message.header.msgh_local_port = client; // локальный порт

// Отправка сообщения
mach_msg(&message.header, // заголовок сообщения
    MACH_SEND_MSG, // отправка сообщения
    sizeof(message), // размер отправленного сообщения
    0, // максимальный размер получаемого сообщения - необязательно
    MACH_PORT_NULL, // имя порта приема - необязательно
    MACH_MSG_TIMEOUT_NONE, // без тайм-аутов
    MACH_PORT_NULL // без уведомлений
);
```

```
    /* Код сервера */

struct message message;

// Получение сообщения
mach_msg(&message.header, // заголовок сообщения
    MACH_RCV_MSG, // получение сообщения
    0, // размер отправленного сообщения
    sizeof(message), // максимальный размер получаемого сообщения
    server, // имя порта приема
    MACH_MSG_TIMEOUT_NONE, // без тайм-аутов
    MACH_PORT_NULL // без уведомлений
);
```

Рисунок 3.18 Пример программы, иллюстрирующей передачу сообщений в Mach.

Сами операции отправки и получения являются гибкими.
Например, когда сообщение отправляется в порт, его очередь может быть заполнена.
Если очередь не заполнена, сообщение копируется в очередь, и задача отправки продолжается.
Если очередь порта заполнена, у отправителя есть несколько вариантов (указанных через параметры mach msg():
1. Ждать бесконечно, пока в очереди не освободится место.
2. Подождите не более n миллисекунд.
3. Не ждите вообще, а сразу возвращайтесь.
4. Временно кэшируйте сообщение.
Здесь сообщение передается операционной системе на сохранение, даже если очередь, в которую это сообщение отправляется, заполнена. Когда сообщение может быть помещено в очередь, отправителю отправляется уведомление. В любой момент времени для данного отправляющего потока может находиться только одно сообщение в полной очереди.
Последний вариант предназначен для серверных задач. После завершения запроса серверной задаче может потребоваться отправить одноразовый ответ задаче, запросившей услугу, но она также должна продолжить обработку других запросов службы, даже если порт ответа для клиента заполнен.

Основной проблемой систем сообщений обычно является низкая производительность, вызванная копированием сообщений из порта отправителя в порт получателя.
Система сообщений Mach пытается избежать операций копирования, используя методы управления виртуальной памятью (глава 10). По сути, Mach отображает адресное пространство, содержащее сообщение отправителя, в адресное пространство получателя.
Следовательно, само сообщение никогда не копируется, поскольку и отправитель, и получатель имеют доступ к одной и той же памяти. Этот метод управления сообщениями обеспечивает значительный прирост производительности, но работает только для внутрисистемных сообщений.

3.7.3 Windows

Операционная система Windows является примером современного дизайна, в котором модульность используется для увеличения функциональности и сокращения времени, необходимого для реализации новых функций. Windows обеспечивает поддержку нескольких

операционных сред или подсистем. Прикладные программы взаимодействуют с этими подсистемами посредством механизма передачи сообщений. Таким образом, прикладные программы можно считать клиентами серверной подсистемы.

Средство передачи сообщений в Windows называется расширенным локальным вызовом процедур (ALPC). Он используется для связи между двумя процессами на одной машине. Он похож на широко используемый стандартный механизм удаленного вызова процедур (RPC), но оптимизирован и специфичен для Windows. (Удаленные вызовы процедур подробно описаны в разделе 3.8.2.) Как и Mach, Windows использует объект порта для установления и поддержания соединения между двумя процессами. Windows использует два типа портов: порты подключения и порты связи. Серверные процессы публикуют объекты портов подключения, которые видны всем процессам. Когда клиенту требуются услуги от подсистемы, он открывает дескриптор объекта порта подключения сервера и отправляет запрос на подключение к этому порту. Затем сервер создает канал и возвращает дескриптор клиенту. Канал состоит из пары частных портов связи: один для сообщений клиент-сервер, другой для сообщений сервер-клиент.
Кроме того, каналы связи поддерживают механизм обратного вызова, который позволяет клиенту и серверу принимать запросы, когда они обычно ожидают ответа.

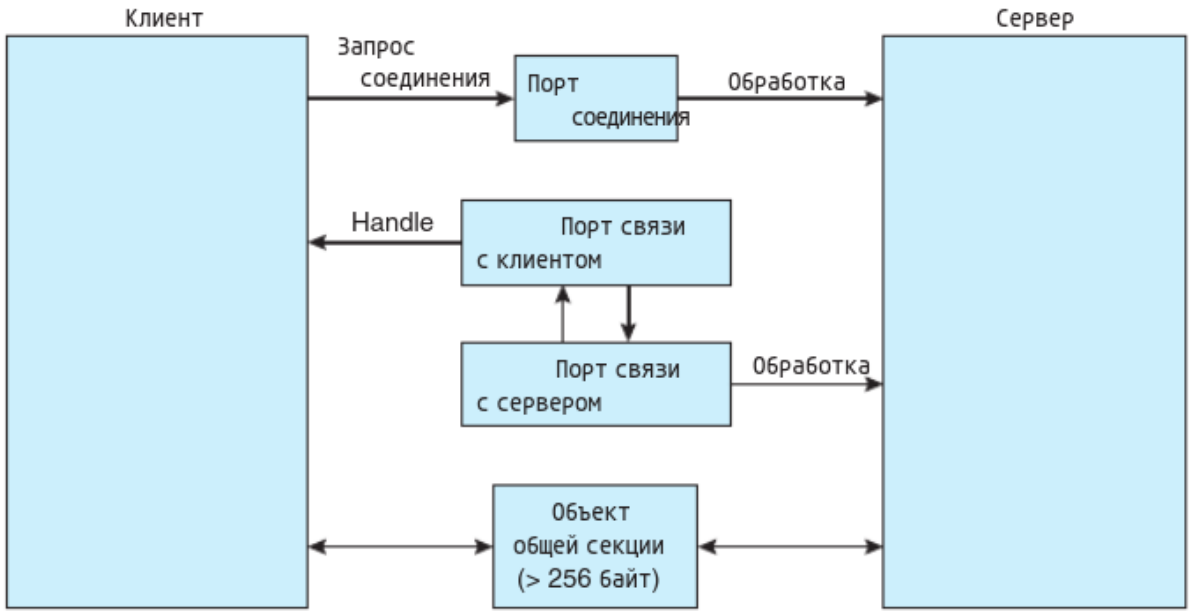


Рисунок 3.19 Расширенные локальные вызовы процедур в Windows.

При создании канала ALPC выбирается один из трех методов передачи сообщений:
1. Для небольших сообщений (до 256 байт) в качестве промежуточного хранилища используется очередь сообщений порта, и сообщения копируются из одного процесса в другой.
2. Сообщения большего размера должны передаваться через объект раздела, который представляет собой область общей памяти, связанную с каналом.
3. Когда объем данных слишком велик, чтобы поместиться в объект раздела, доступен API, который позволяет серверным процессам читать и записывать непосредственно в адресное пространство клиента.

Клиент должен решить при настройке канала, нужно ли ему отправлять большое сообщение. Если клиент определяет, что он действительно хочет отправлять большие сообщения, он запрашивает создание объекта раздела. Аналогично, если сервер решает, что ответы будут большими, он создает объект раздела. Чтобы объект раздела можно было использовать, отправляется небольшое сообщение, содержащее указатель и информацию о размере объекта раздела. Этот метод более сложен, чем первый из перечисленных выше, но позволяет избежать копирования данных. Структура расширенных вызовов локальных процедур в Windows показана на рисунке 3.19.

Важно отметить, что функция ALPC в Windows не является частью Windows API и, следовательно, не видна программисту приложения. Вместо этого приложения, использующие Windows API, вызывают стандартные удаленные вызовы процедур. Когда RPC вызывается для процесса в той же системе, RPC обрабатывается косвенно через вызов процедуры ALPC. Кроме того, многие службы ядра используют ALPC для взаимодействия с клиентскими процессами.

3.7.4 Каналы (pipes)

Канал действует как кабель, позволяющий взаимодействовать двум процессам. Каналы были одним из первых механизмов IPC в ранних системах UNIX. Обычно они предоставляют один из самых простых способов взаимодействия процессов друг с другом, хотя у них также есть некоторые ограничения. При реализации канала необходимо учитывать четыре вопроса:

1. Обеспечивает ли канал двустороннюю связь или связь однонаправленная?
2. Если разрешена двусторонняя связь, является ли она полудуплексной (данные могут передаваться только в одном направлении одновременно) или полнодуплексной (данные могут передаваться в обоих направлениях одновременно)?
3. Должны ли существовать отношения (например, родитель-ребенок) между коммуникативными процессами?
4. Могут ли каналы обмениваться данными по сети или взаимодействующие процессы должны находиться на одной машине?

В следующих разделах мы рассмотрим два распространенных типа каналов, используемых как в системах UNIX, так и в Windows: обычные каналы и именованные каналы.

3.7.4.1 Обычные каналы

Обычные каналы позволяют двум процессам взаимодействовать стандартным образом производитель-потребитель: производитель записывает на один конец канала (конец записи), а потребитель читает с другого конца (конец чтения). В результате обычные каналы являются однонаправленными, обеспечивая только одностороннюю связь. Если требуется двусторонняя связь, необходимо использовать два канала, каждый из которых отправляет данные в разном направлении. Далее мы проиллюстрируем построение обычных каналов в системах UNIX и Windows. В обоих примерах программы один процесс записывает сообщение Приветствие в канал, а другой процесс читает это сообщение из канала.

В системах UNIX обычные каналы создаются с помощью функции

```
pipe(int fd[])
```

Эта функция создает канал, доступ к которому осуществляется через файловые дескрипторы int fd[]: fd[0] — конец канала для чтения, а fd[1] — конец записи. UNIX рассматривает канал как

файл особого типа. Таким образом, доступ к каналам можно получить с помощью обычных системных вызовов read() и write(). К обычному каналу невозможно получить доступ извне процесса, который его создал. Обычно родительский процесс создает канал и использует его для связи с дочерним процессом, который он создает с помощью fork(). Напомним из раздела 3.3.1, что дочерний процесс наследует открытые файлы от своего родителя. Поскольку канал представляет собой файл особого типа, дочерний процесс наследует канал от родительского процесса.

Рисунок 3.20 иллюстрирует связь файловых дескрипторов в массиве fd с родительским и дочерним процессами.

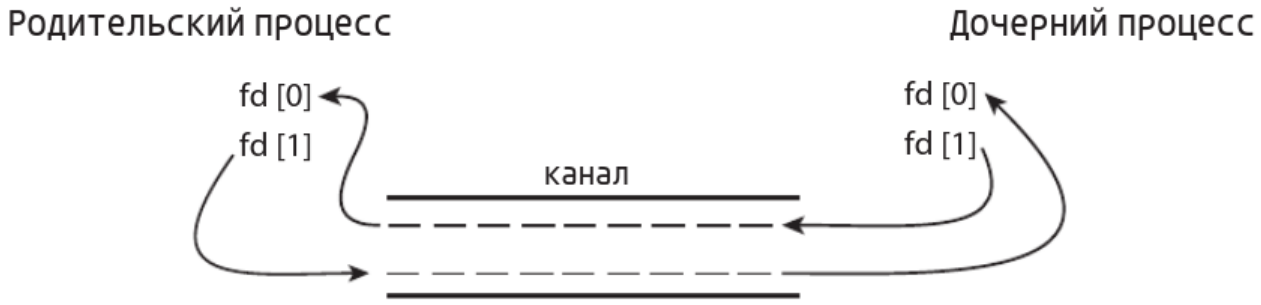


Рисунок 3.20 Дескрипторы файлов для обычного канала.

Как это показано, любые записи родительского элемента на его конец записи канала — fd[1] — могут быть прочитаны дочерним элементом со своего конца чтения — fd[0] — канала.

```
#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#define BUFFER SIZE 25
#define READ END 0
#define WRITE END 1

int main(void)
{
    char write msg[BUFFER SIZE] = "Greetings";
    char read msg[BUFFER SIZE];
    int fd[2];
    pid t pid;

        /* Продолжение программы на Рисунке 3.22 */
```

Рисунок 3.21 Обычный канал в UNIX.

В программе UNIX, показанной на рис. 3.21, родительский процесс создает канал, а затем отправляет вызов fork(), создавая дочерний процесс. Что происходит после вызова fork(), зависит от того, как данные будут проходить через канал. В этом случае родительский элемент

записывает в канал, а дочерний элемент читает из него. Важно отметить, что как родительский процесс, так и дочерний процесс изначально закрывают неиспользуемые концы канала.
Хотя программа, показанная на рисунке 3.21, не требует этого действия, это важный шаг, чтобы гарантировать, что процесс чтения из канала может обнаружить конец файла (read() возвращает 0), когда записывающий модуль закрыл свой конец канала.

Обычные каналы в системах Windows называются анонимными каналами и ведут себя аналогично своим аналогам в UNIX: они однонаправлены и используют отношения «родитель-потомок» между взаимодействующими процессами. Кроме того, чтение и запись в канал можно выполнить с помощью обычных функций ReadFile() и WriteFile().
Windows API для создания каналов — это функция CreatePipe(), которой передаются четыре параметра. Параметры предоставляют отдельные дескрипторы для (1) чтения и (2) записи в канал, а также (3) экземпляр структуры STARTUPINFO, которая используется для указания того, что дочерний процесс должен наследовать дескрипторы канала. Кроме того, (4) может быть указан размер канала (в байтах).

Рисунок 3.23 иллюстрирует родительский процесс, создающий анонимный канал для связи со своим дочерним процессом. В отличие от систем UNIX, в которых дочерний процесс автоматически наследует канал, созданный его родителем, Windows требует от программиста указать, какие атрибуты унаследует дочерний процесс. Это достигается путем первой инициализации структуры SECURITY_ATTRIBUTES, позволяющей наследовать дескрипторы, а затем перенаправления дескрипторов дочернего процесса для стандартного ввода или стандартного вывода на дескриптор чтения или записи канала. Поскольку дочерний элемент будет читать из канала, родительский элемент должен перенаправить стандартный ввод дочернего элемента в дескриптор чтения канала. Кроме того, поскольку каналы являются полудуплексными, необходимо запретить дочернему элементу наследовать конец канала для записи. Программа создания дочернего процесса аналогична программе на рис. 3.10, за исключением того, что пятый параметр имеет значение TRUE, что указывает на то, что дочерний процесс должен наследовать назначенные дескрипторы от своего родителя. Перед записью в канал родительский элемент сначала закрывает неиспользуемый для чтения конец канала.
Дочерний процесс, считывающий данные из канала, показан на рисунке 3.25. Перед чтением из канала эта программа получает дескриптор чтения канала, вызывая GetStdHandle().

```
/* create the pipe */
    if (pipe(fd) == -1) {
        fprintf(stderr,"Pipe failed");
        return 1;
    }
    /* fork a child process */
    pid = fork();
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }

    if (pid > 0) { /* parent process */
        /* close the unused end of the pipe */
        close(fd[READ END]);

        /* write to the pipe */
        write(fd[WRITE END], write msg, strlen(write msg)+1);

        /* close the write end of the pipe */
```

```
        close(fd[WRITE END]);
    }
    else { /* child process */
        /* close the unused end of the pipe */
        close(fd[WRITE END]);
        /* read from the pipe */
        read(fd[READ END], read msg, BUFFER SIZE);
        printf("read %s",read msg);
        /* close the read end of the pipe */
        close(fd[READ END]);
    }
    return 0;
}
```

Рисунок 3.22 Рисунок 3.21, продолжение

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>

#define BUFFER SIZE 25

int main(VOID)
{
    HANDLE ReadHandle, WriteHandle;
    STARTUPINFO si;
    ROCESS INFORMATION pi;
    char message[BUFFER SIZE] = "Greetings";
    DWORD written;

        /* Программа продолжается на рисунке 3.24 */
```

Рисунок 3.23. Анонимный канал Windows — родительский процесс.

Обратите внимание, что обычные каналы требуют отношений «родитель-потомок» между взаимодействующими процессами как в системах UNIX, так и в системах Windows. Это означает, что эти каналы можно использовать только для связи между процессами на одной машине.

3.7.4.2 Именованные каналы

Обычные каналы предоставляют простой механизм, позволяющий взаимодействовать паре процессов. Однако обычные каналы существуют только пока процессы общаются друг с другом. Как в системах UNIX, так и в Windows, как только процессы завершают взаимодействие и завершаются, обычный канал перестает существовать.

Именованные каналы предоставляют гораздо более мощный инструмент связи. Коммуникация может быть двунаправленной, и отношения родитель-потомок не требуются. Как только именованный канал установлен, несколько процессов могут использовать его для связи. Фактически, в типичном сценарии именованный канал имеет несколько авторов. Кроме того, именованные каналы продолжают существовать после завершения процессов связи. Системы UNIX и Windows поддерживают именованные каналы, хотя детали реализации сильно различаются. Далее мы исследуем именованные каналы в каждой из этих систем.

Именованные каналы в системах UNIX называются FIFO. После создания они выглядят как обычные файлы в файловой системе. FIFO создается с помощью системного вызова mkfifo() и управляется с помощью обычных системных вызовов open(), read(), write() и close(). Он будет продолжать существовать до тех пор, пока не будет явно удален из файловой системы.
Хотя FIFO допускают двунаправленную связь, допускается только полудуплексная передача. Если данные должны передаваться в обоих направлениях, обычно используются два FIFO. Кроме того, взаимодействующие процессы должны находиться на одной машине.
Если требуется межмашинная связь, необходимо использовать сокеты (раздел 3.8.1).

```
/* Установка атрибутов безопасности, позволяющих наследовать каналы */
SECURITY_ATTRIBUTES sa = {sizeof(SECURITY_ATTRIBUTES),NULL,TRUE};
/* Выделение памяти */
ZeroMemory(&pi, sizeof(pi));
/* Создание канала */

if (!CreatePipe(&ReadHandle, &WriteHandle, &sa, 0)) {
    fprintf(stderr, "Create Pipe Failed"); // Ошибка при создании канала
    return 1;
}

/* Настройка структуры STARTUPINFO для дочернего процесса */
GetStartupInfo(&si);
si.hStdOutput = GetStdHandle(STD_OUTPUT_HANDLE);

/* Перенаправление стандартного ввода на чтение конца канала */
si.hStdInput = ReadHandle;
si.dwFlags = STARTF_USESTDHANDLES;

/* Не разрешать дочернему процессу наследовать записывающий конец канала */
SetHandleInformation(WriteHandle, HANDLE_FLAG_INHERIT, 0);

/* Создание дочернего процесса */
CreateProcess(NULL, "child.exe", NULL, NULL,
    TRUE, /* наследование дескрипторов */
    0, NULL, NULL, &si, &pi);

/* Закрыть неиспользуемый конец канала */
CloseHandle(ReadHandle);

/* Родитель пишет в канал */
if (!WriteFile(WriteHandle, message, BUFFER_SIZE, &written, NULL))
    fprintf(stderr, "Error writing to pipe."); // Ошибка при записи в канал

/* Закрыть записывающий конец канала */
CloseHandle(WriteHandle);

/* Ожидание завершения дочернего процесса */
WaitForSingleObject(pi.hProcess, INFINITE);
CloseHandle(pi.hProcess);
CloseHandle(pi.hThread);
return 0;
}
```

Рисунок 3.24 Рисунок 3.23, продолжение.

```
#include <stdio.h>
#include <windows.h>

#define BUFFER_SIZE 25

int main(VOID)
{
HANDLE ReadHandle;
CHAR buffer[BUFFER_SIZE];
DWORD read;

    /* Получение дескриптора чтения канала */
    ReadHandle = GetStdHandle(STD_INPUT_HANDLE);

    /* Дочерний процесс читает из канала */
    if (ReadFile(ReadHandle, buffer, BUFFER_SIZE, &read, NULL))
        printf("child read %s",buffer); // Дочерний процесс читает из канала
    else
        fprintf(stderr, "Error reading from pipe"); // Ошибка при чтении из канала
    return 0;
}
```

Рисунок 3.25 Анонимные каналы Windows — дочерний процесс.

Именованные каналы в системах Windows предоставляют более богатый механизм связи, чем их аналоги в UNIX. Разрешена полнодуплексная связь, и процессы связи могут находиться как на одной, так и на разных машинах. Кроме того, через UNIX FIFO могут передаваться только байтовые данные, тогда как системы Windows допускают передачу данных как в байтах, так и в сообщениях. Именованные каналы создаются с помощью функции CreateNamedPipe(), и клиент может подключиться к именованному каналу с помощью ConnectNamedPipe().
Связь через именованный канал можно осуществить с помощью функций ReadFile() и WriteFile().

3.8 Связь в клиент-серверных системах

В разделе 3.4 мы описали, как процессы могут взаимодействовать, используя общую память и передачу сообщений. Эти методы также можно использовать для связи в системах клиент-сервер (раздел 1.10.3). В этом разделе мы исследуем две другие стратегии взаимодействия в системах клиент-сервер: сокеты и удаленные вызовы процедур (RPC). Как мы увидим в нашем обзоре RPC, они не только полезны для клиент-серверных вычислений, но Android также использует удаленные процедуры как форму IPC между процессами, работающими в одной системе.

3.8.1 Сокеты

Сокет определяется как конечная точка для связи. Пара процессов, взаимодействующих по сети, использует пару сокетов — по одному для каждого процесса. Сокет идентифицируется IP-адресом, объединенным с номером порта. В общем, сокеты используют архитектуру клиент-сервер. Сервер ожидает входящие клиентские запросы, прослушивая указанный порт. После получения запроса сервер принимает соединение из клиентского сокета для завершения соединения. Серверы, реализующие определенные службы (такие как SSH, FTP и HTTP), прослушивают общеизвестные порты (сервер SSH прослушивает порт 22; сервер FTP прослушивает порт 21; веб-сервер или HTTP-сервер прослушивает порт 80). .
Все порты ниже 1024 считаются хорошо известными и используются для реализации стандартных сервисов.

Когда клиентский процесс инициирует запрос на соединение, хост-компьютер назначает ему порт. Этот порт имеет произвольное число больше 1024. Например, если клиент на хосте X с IP-адресом 146.86.5.20 желает установить соединение с веб-сервером (который прослушивает порт 80) по адресу 161.25.19.8, хосту X может быть назначен порт 1625. Соединение будет состоять из пары сокетов: (146.86.5.20:1625) на хосте X и (161.25.19.8:80) на веб-сервере. Эта ситуация проиллюстрирована на рисунке 3.26. Пакеты, перемещающиеся между хостами, доставляются соответствующему процессу в зависимости от номера порта назначения.

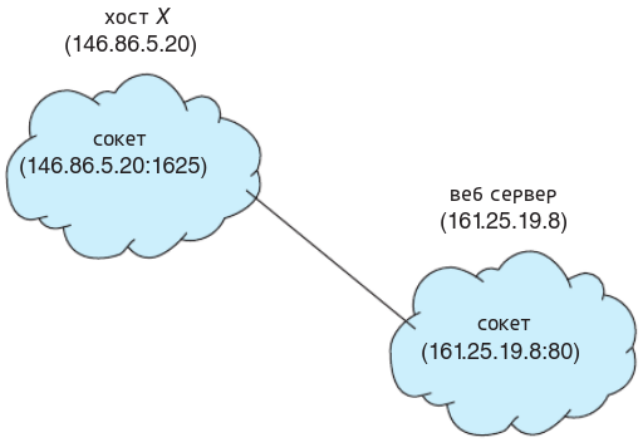


Рисунок 3.26 Общение с помощью сокетов.

Все соединения должны быть уникальными. Следовательно, если другой процесс также на хосте X захочет установить другое соединение с тем же веб-сервером, ему будет присвоен номер порта больше 1024, а не равный 1625. Это гарантирует, что все соединения состоят из уникальной пары сокетов.

Хотя в большинстве примеров программ в этом тексте используется C, сокеты мы будем иллюстрировать с помощью Java, поскольку он обеспечивает гораздо более простой интерфейс для сокетов и имеет богатую библиотеку сетевых утилит. Тем, кто интересуется программированием сокетов на C или C++, следует обратиться к библиографическим примечаниям в конце главы.

Java предоставляет три различных типа сокетов. Сокеты с установлением соединения (TCP) реализуются с помощью класса Socket. Сокеты без установления соединения (UDP) используют класс DatagramSocket. Наконец, класс MulticastSocket является подклассом класса DatagramSocket. Многоадресный сокет позволяет отправлять данные нескольким получателям. В нашем примере описывается сервер данных, использующий TCP-сокеты с установлением соединения. Эта операция позволяет клиентам запрашивать текущую дату и время с сервера. Сервер прослушивает порт 6013, хотя порт может иметь любой произвольный неиспользуемый номер, превышающий 1024. При получении соединения сервер возвращает клиенту дату и время.

Сервер данных показан на рисунке 3.27. Сервер создает ServerSocket, который указывает, что он будет прослушивать порт 6013. Затем сервер начинает прослушивать порт с помощью метода Accept(). Сервер блокирует метод Accept(), ожидая, пока клиент запросит соединение. Когда получен запрос на соединение, метод Accept() возвращает сокет, который сервер может использовать для связи с клиентом.

Подробности того, как сервер взаимодействует с сокетом, следующие. Сервер сначала устанавливает объект PrintWriter, который он будет использовать для связи с клиентом. Объект PrintWriter позволяет серверу выполнять запись в сокет, используя для вывода обычные методы print() и println(). Серверный процесс отправляет дату клиенту, вызывая метод println(). После записи даты в сокет сервер закрывает сокет для клиента и возобновляет прослушивание новых запросов.

```
import java.net.*;
import java.io.*;

public class DateServer
{
    public static void main(String[] args) {
        try {
            ServerSocket sock = new ServerSocket(6013);

            /* now listen for connections */
            while (true) {
                Socket client = sock.accept();

                PrintWriter pout = new
                    PrintWriter(client.getOutputStream(), true);
                /* write the Date to the socket */
                pout.println(new java.util.Date().toString());

                /* close the socket and resume */
                /* listening for connections */
                client.close();
            }
        }
        catch (IOException ioe) {
            System.err.println(ioe);
        }
    }
}
```

Рисунок 3.27 Сервер данных.

Клиент взаимодействует с сервером, создавая сокет и подключаясь к порту, который прослушивает сервер. Мы реализуем такой клиент в программе Java, показанной на рисунке 3.28. Клиент создает сокет и запрашивает соединение с сервером по IP-адресу 127.0.0.1 через порт 6013. После установления соединения клиент может читать данные из сокета, используя обычные операторы потокового ввода-вывода. После получения даты от сервера клиент закрывает сокет и завершает работу. IP-адрес 127.0.0.1 — это специальный IP-адрес, известный как петля. Когда компьютер обращается к IP-адресу 127.0.0.1, он ссылается на себя.
Этот механизм позволяет клиенту и серверу на одном хосте взаимодействовать с использованием протокола TCP/IP. IP-адрес 127.0.0.1 можно заменить IP-адресом другого хоста, на котором работает сервер данных. Помимо IP-адреса можно также использовать фактическое имя хоста, например www.westminstercollege.edu.

```
import java.net.*;
import java.io.*;

public class DateClient {
   public static void main(String[] args) {
      try {
         /* Устанавливаем соединение с серверным сокетом */
         Socket sock = new Socket("127.0.0.1", 6013);
         InputStream in = sock.getInputStream();
         BufferedReader bin = new BufferedReader(new InputStreamReader(in));
         /* Читаем дату из сокета */
         String line;
         while ((line = bin.readLine()) != null)
            System.out.println(line);
         /* Закрываем соединение с сокетом */
         sock.close();
      } catch (IOException ioe) {
         System.err.println(ioe);
      }
   }
}
```

Рисунок 3.28 Клиент даты.

Связь с использованием сокетов, хотя она и распространена и эффективна, считается низкоуровневой формой связи между распределенными процессами. Одна из причин заключается в том, что сокеты позволяют обмениваться между взаимодействующими потоками только неструктурированным потоком байтов. Ответственность за определение структуры данных лежит на клиентском или серверном приложении. В следующем подразделе мы рассмотрим метод связи более высокого уровня: удаленные вызовы процедур (RPC).

3.8.2 Удаленные вызовы процедур

Одной из наиболее распространенных форм удаленного обслуживания является парадигма RPC, которая была разработана как способ абстрагировать механизм вызова процедур для

использования между системами с сетевыми соединениями. Он во многом похож на механизм IPC, описанный в разделе 3.4, и обычно строится поверх такой системы. Однако здесь, поскольку мы имеем дело со средой, в которой процессы выполняются в отдельных системах, мы должны использовать схему связи на основе сообщений для предоставления удаленного обслуживания.

В отличие от сообщений IPC, сообщения, которыми обмениваются при связи RPC, хорошо структурированы и, таким образом, больше не являются просто пакетами данных. Каждое сообщение адресовано демону RPC, прослушивающему порт удаленной системы, и каждое сообщение содержит идентификатор, определяющий функцию, которую необходимо выполнить, и параметры, которые необходимо передать этой функции.
Затем функция выполняется по запросу, и любые выходные данные отправляются обратно запрашивающей стороне в отдельном сообщении.

Порт в этом контексте — это просто номер, включенный в начало пакета сообщения. Хотя система обычно имеет один сетевой адрес, в этом адресе может быть множество портов, позволяющих различать множество поддерживаемых ею сетевых служб. Если удаленному процессу требуется услуга, он направляет сообщение на соответствующий порт. Например, если система хочет, чтобы другие системы могли составлять список своих текущих пользователей, у нее должен быть демон, поддерживающий такой RPC, подключенный к порту, скажем, к порту 3027. Любая удаленная система может получить необходимую информацию (то есть список текущих пользователей), отправив сообщение RPC на порт 3027 сервера. Данные будут получены в ответном сообщении.

Семантика RPC позволяет клиенту вызывать процедуру на удаленном хосте так же, как он вызывал бы процедуру локально. Система RPC скрывает детали, позволяющие осуществлять связь, предоставляя заглушку на стороне клиента. Обычно для каждой отдельной удаленной процедуры существует отдельная заглушка. Когда клиент вызывает удаленную процедуру, система RPC вызывает соответствующую заглушку, передавая ей параметры, предоставленные удаленной процедуре. Эта заглушка находит порт на сервере и маршалирует параметры.
Затем заглушка передает сообщение на сервер, используя передачу сообщений.
Аналогичная заглушка на стороне сервера получает это сообщение и вызывает процедуру на сервере. При необходимости возвращаемые значения передаются обратно клиенту с использованием того же метода. В системах Windows код-заглушка компилируется на основе спецификации, написанной на языке Microsoft Interface Definitio (MIDL), который используется для определения интерфейсов между клиентскими и серверными программами.

Маршалинг параметров решает проблему различий в представлении данных на клиентских и серверных компьютерах. Рассмотрим представление 32-битных целых чисел. Некоторые системы (так называемые «с прямым порядком байтов») сначала сохраняют наиболее значимый байт, тогда как другие системы (известные как «с прямым порядком байтов») сначала сохраняют младший байт. Ни один из этих порядков не является «лучшим» сам по себе; скорее, выбор произволен в рамках компьютерной архитектуры. Чтобы устранить подобные различия, многие системы RPC определяют машинно-независимое представление данных.
Одно из таких представлений известно как представление внешних данных (XDR). На стороне клиента маршалинг параметров включает преобразование машинно-зависимых данных в XDR перед их отправкой на сервер. На стороне сервера данные XDR демаршализуются и преобразуются в машинно-зависимое представление для сервера.

Другой важный вопрос касается семантики вызова. В то время как вызовы локальных процедур завершаются неудачно только в чрезвычайных обстоятельствах, RPC могут давать сбой или дублироваться и выполняться более одного раза в результате типичных сетевых ошибок.

Один из способов решения этой проблемы — обеспечить, чтобы операционная система обрабатывала сообщения ровно один раз, а не не более одного раза. Большинство вызовов локальных процедур имеют функцию «ровно один раз», но ее сложнее реализовать.

Во-первых, рассмотрим «не более одного раза». Эту семантику можно реализовать путем прикрепления временной метки к каждому сообщению. Сервер должен хранить историю всех временных меток сообщений, которые он уже обработал, или историю, достаточно большую, чтобы гарантировать обнаружение повторяющихся сообщений. Входящие сообщения, метка времени которых уже есть в истории, игнорируются. Затем клиент может отправить сообщение один или несколько раз и быть уверенным, что оно будет выполнено только один раз.

Для «ровно один раз» нам нужно устранить риск того, что сервер никогда не получит запрос. Для этого сервер должен реализовать протокол «не более одного раза», описанный выше, но также должен подтвердить клиенту, что вызов RPC был получен и выполнен.
Эти сообщения ACK являются общими для всей сети. Клиент должен периодически повторно отправлять каждый вызов RPC, пока не получит подтверждение для этого вызова.
Еще один важный вопрос касается связи между сервером и клиентом. При стандартных вызовах процедур во время компоновки, загрузки или выполнения происходит определенная форма связывания (глава 9), так что имя вызова процедуры заменяется адресом памяти, в котором она была вызвана. Схема RPC требует аналогичной привязки клиента и порта сервера, но откуда клиент узнает номера портов на сервере? Ни одна из систем не имеет полной информации о другой, поскольку они не используют общую память.

Распространены два подхода. Во-первых, информация привязки может быть заранее определена в форме фиксированных адресов портов. Во время компиляции с вызовом RPC связан фиксированный номер порта. После компиляции программы сервер не может изменить номер порта запрошенной службы. Во-вторых, привязка может выполняться динамически с помощью механизма рандеву. Обычно операционная система предоставляет демон рандеву (также называемый «сопоставителем») на фиксированном порту RPC.
Затем клиент отправляет сообщение, содержащее имя RPC, демону рандеву, запрашивая адрес порта RPC, который ему необходимо выполнить. Возвращается номер порта, и вызовы RPC могут отправляться на этот порт до тех пор, пока процесс не завершится (или не произойдет сбой сервера). Этот метод требует дополнительных затрат на первоначальный запрос, но он более гибок, чем первый подход. На рис. 3.29 показан пример взаимодействия.

Схема RPC полезна при реализации распределенной файловой системы (глава 19). Такая система может быть реализована как набор демонов и клиентов RPC. Сообщения адресуются порту распределенной файловой системы на сервере, на котором должна выполняться файловая операция. Сообщение содержит операцию с диском, которую необходимо выполнить.
Дисковая операция может быть read(), write(), rename(), delete() или status(), что соответствует обычным системным вызовам, связанным с файлами. Возвращаемое сообщение содержит все данные, полученные в результате этого вызова, который выполняется демоном DFS от имени клиента. Например, сообщение может содержать запрос на передачу клиенту всего файла или ограничиваться простым запросом блокировки. В последнем случае может потребоваться несколько запросов, если необходимо передать целый файл.

3.8.2.1 Android RPC

Хотя RPC обычно связаны с клиент-серверными вычислениями в распределенной системе, их также можно использовать как форму IPC между процессами, работающими в одной системе. Операционная система Android имеет богатый набор механизмов IPC, содержащихся в ее связующей структуре, включая RPC, которые позволяют одному процессу запрашивать услуги у другого процесса.

Android определяет компонент приложения как базовый строительный блок, который обеспечивает полезность приложения Android, и приложение может объединять несколько компонентов приложения для обеспечения функциональности приложения.
Одним из таких компонентов приложения является служба, которая не имеет пользовательского интерфейса, а работает в фоновом режиме при выполнении длительных операций или выполнении работы для удаленных процессов.

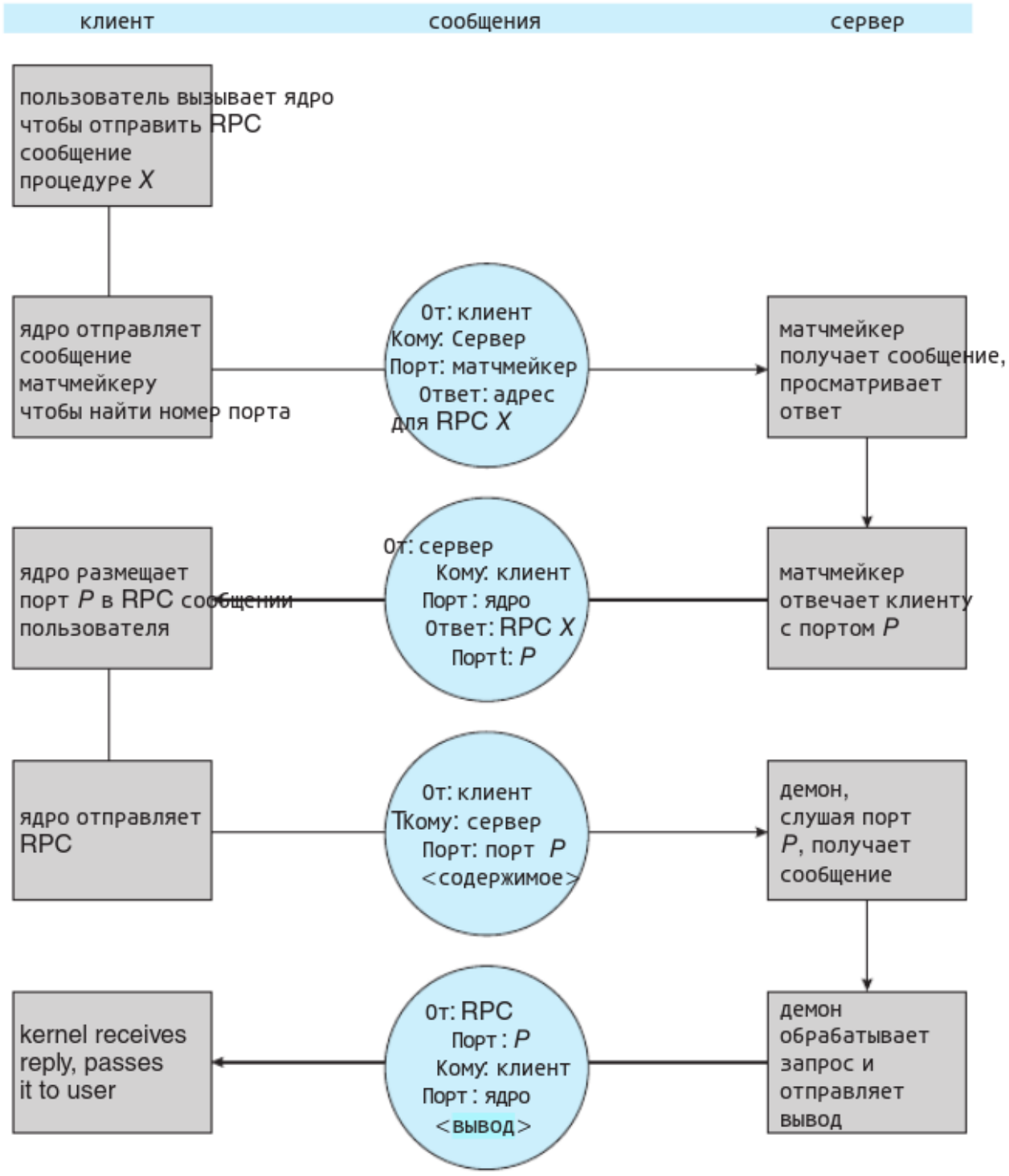


Рисунок 3.29 Выполнение удаленного вызова процедур (RPC).

Примеры служб включают воспроизведение музыки в фоновом режиме и получение данных через сетевое соединение от имени другого процесса, тем самым предотвращая блокировку другого процесса во время загрузки данных.

Когда клиентское приложение вызывает метод службыbindService(), эта служба «привязывается» и доступна для обеспечения связи клиент-сервер с использованием либо передачи сообщений, либо RPC.

Связанная служба должна расширять класс Android Service и должна реализовывать метод onBind(), который вызывается, когда клиент вызывает метод bindService(). В случае передачи сообщений метод onBind() возвращает службу сообщений, которая используется для отправки сообщений от клиента в службу. Служба обмена сообщениями работает только в одну сторону; если служба должна отправить ответ обратно клиенту, клиент также должен предоставить службу обмена сообщениями, которая содержится в поле ответить объекта Message, отправленного в службу. Затем служба может отправлять сообщения обратно клиенту.

Чтобы предоставить RPC, метод onBind() должен возвращать интерфейс, представляющий методы удаленного объекта, которые клиенты используют для взаимодействия со службой. Этот интерфейс написан на обычном синтаксисе Java и использует язык определения интерфейса Android (AIDL) для создания файлов-заглушек, которые служат клиентским интерфейсом для удаленных служб.

Здесь мы кратко описываем процесс, необходимый для предоставления универсальной удаленной службы с именем RemoteMethod() с использованием AIDL и службы связывания. Интерфейс удаленного сервиса выглядит следующим образом:

```
/* RemoteService.aidl */
interface RemoteService
{
    boolean remoteMethod(int x, double y);
}
```

Этот файл записывается как RemoteService.aidl. Комплект разработки Android будет использовать его для создания интерфейса .java из файла .aidl файл, а также заглушку, которая служит интерфейсом RPC для этой службы. Сервер должен реализовать интерфейс, созданный файлом .aidl, и реализация этого интерфейса будет вызываться, когда клиент вызывает RemoteMethod(). Когда клиент вызывает метод bindService(), на сервере вызывается метод onBind(), который возвращает клиенту заглушку для объекта RemoteService.
Затем клиент может вызвать удаленный метод следующим образом:

```
RemoteService service;
...
service.remoteMethod(3, 0.14);
```

На внутреннем уровне платформа связывания Android обрабатывает маршалинг параметров, передает маршалированные параметры между процессами и вызывает необходимую реализацию службы, а также отправляет любые возвращаемые значения обратно клиентскому процессу.

3.9 Резюме

• Процесс — это выполняемая программа, а состояние текущей активности процесса представлено программным счетчиком, а также другими регистрами.
• Структура процесса в памяти представлена четырьмя различными разделами: (1) текст, (2) данные, (3) куча и (4) стек.
• По мере выполнения процесса он меняет состояние. Существует четыре основных состояния процесса: (1) готовность, (2) выполнение, (3) ожидание и (4) завершение.

• Блок управления процессом (PCB) — это структура данных ядра, которая представляет процесс в операционной системе.
• Роль планировщика процессов заключается в выборе доступного процесса для запуска на ЦП.
• Операционная система выполняет переключение контекста, когда переключается с запуска одного процесса на запуск другого.
• Системные вызовы fork() и CreateProcess() используются для создания процессов в системах UNIX и Windows соответственно.
• Когда для связи между процессами используется общая память, два (или более) процесса используют одну и ту же область памяти. POSIX предоставляет API для общей памяти.
• Два процесса могут взаимодействовать посредством обмена сообщениями друг с другом с помощью передачи сообщений. Операционная система Mach использует передачу сообщений в качестве основной формы межпроцессного взаимодействия. Windows также предоставляет форму передачи сообщений.
• Канал обеспечивает канал для взаимодействия двух процессов. Существует две формы каналоканалов: обычные и именные. Обычные каналы предназначены для связи между процессами, имеющими отношения родитель-потомок. Именованные каналы более общие и позволяют взаимодействовать нескольким процессам.
• Системы UNIX предоставляют обычные каналы посредством системного вызова Pipe(). Обычные каналы имеют конец чтения и конец записи. Родительский процесс может, например, отправлять данные в канал, используя свою сторону записи, а дочерний процесс может читать их со своей стороны чтения. Именованные каналы в UNIX называются FIFO.
• Системы Windows также предоставляют две формы каналов — анонимные и именованные каналы. Анонимные каналы аналогичны обычным каналам UNIX. Они однонаправлены и используют отношения родитель-потомок между коммуникативными процессами. Именованные каналы предлагают более богатую форму межпроцессного взаимодействия, чем аналог UNIX — FIFO.
• Двумя распространенными формами взаимодействия клиент-сервер являются сокеты и удаленные вызовы процедур (RPC). Сокеты позволяют двум процессам на разных машинах взаимодействовать по сети. RPC абстрагируют концепцию вызовов функций (процедур) таким образом, что функцию можно вызвать в другом процессе, который может находиться на отдельном компьютере.
• Операционная система Android использует RPC как форму межпроцессного взаимодействия, используя свою структуру связывания.

Практические упражнения

3.1 Используя программу, показанную на рисунке 3.30, объясните, каким будет выходной сигнал на ЛИНИИ A.
3.2 Сколько процессов создается программой, показанной на рисунке 3.31, включая исходный родительский процесс?
3.3 Исходные версии мобильной операционной системы Apple iOS не предусматривали средств параллельной обработки.
Обсудите три основные сложности, которые параллельная обработка добавляет в операционную систему.
3.4 Некоторые компьютерные системы предоставляют несколько наборов регистров.
Опишите, что происходит, когда происходит переключение контекста, если новый контекст уже загружен в один из наборов регистров.
Что произойдет, если новый контекст находится в памяти, а не в наборе регистров, и все наборы регистров используются?

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int value = 5;

int main() {
    pid_t pid;

    pid = fork();

    if (pid == 0) { /* дочерний процесс */
        value += 15;
        return 0;
    } else if (pid > 0) { /* родительский процесс */
        wait(NULL);
        printf("PARENT: value = %d", value); /* СТРОКА A */
        return 0;
    }
}
```

Рисунок 3.30. Какой выход будет на строке A?

3.5 Когда процесс создает новый процесс с помощью операции fork(), какое из следующих состояний является общим для родительского процесса и дочернего процесса?
а. Стек
б. Куча
в. Сегменты общей памяти

3.6. Рассмотрим семантику «ровно один раз» в отношении механизма RPC.
Правильно ли выполняется алгоритм реализации этой семантики, даже если сообщение ACK, отправленное обратно клиенту, потеряно из-за сетевой проблемы? Опишите последовательность сообщений и обсудите, сохраняется ли еще «ровно один раз».
3.7 Предположим, что распределенная система подвержена сбою сервера. Какие механизмы потребуются, чтобы гарантировать семантику «ровно один раз» для выполнения RPC?

```
#include <stdio.h>
#include <unistd.h>

int main() {
    /* Создаем дочерний процесс */
    fork();

    /* Создаем еще один дочерний процесс */
    fork();

    /* И еще один */
    fork();

    return 0;
}
```

Рисунок 3.31 Сколько процессов создано?

Дальнейшее чтение

Создание процессов, управление ими и IPC в системах UNIX и Windows соответственно обсуждаются в [Роббинс и Роббинс (2003)] и [Русинович и др. (2017)].
[Love (2010)] описывает поддержку процессов в ядре Linux, а [Hart (2005)] подробно описывает системное программирование Windows.
Описание многопроцессной модели, используемой в Google Chrome, можно найти по адресу http://blog.chromium.org/2008/09/multi-process-architecture.html.
Передача сообщений в многоядерных системах обсуждается в [Holland and Seltzer (2011)].
[Левин (2013)] описывает передачу сообщений в системе Mach, особенно в отношении Mac OS и i OS.
[Гарольд (2005)] освещает программирование сокетов на Java.
Подробную информацию об Android RPC можно найти по адресу https://developer.android.com/guide/comComponents/aidl.html.
[Hart (2005)] и [Robbins and Robbins (2003)] описывают каналы в системах Windows и UNIX соответственно.
Рекомендации по разработке для Android можно найти по адресу https://developer.android.com/guide/.

Библиографический список

[Гарольд (2005)] Э. Р. Гарольд, Сетевое программирование на Java, третье издание, O’Reilly & Associates (2005).
[Харт (2005)] Дж. М. Харт, Системное программирование для Windows, третье издание, Аддисон-Уэсли (2005).
[Холланд и Зельцер (2011)] Д. Холланд и М. Зельцер, «Многоядерные операционные системы: взгляд в будущее с 1991 г., э., 2011 г.», Материалы 13-й конференции USENIX по «Горячим темам в операционных системах» (2011 г.), стр. 33–33 .
[Левин (2013)] Дж. Левин, Внутреннее устройство Mac OS X и iOS до ядра Apple, Wiley (2013).
[Лав (2010)] Р. Лав, Разработка ядра Linux, третье издание, Библиотека разработчика (2010).
[Роббинс и Роббинс (2003)] К. Роббинс и С. Роббинс, Программирование систем Unix: связь, параллелизм и потоки, второе издание, Prentice Hall (2003).
[Русинович и др. (2017)] М. Руссинович, Д. А. Соломон и А. Ионеску, Внутреннее устройство Windows – Часть 1, седьмое издание, Microsoft Press (2017).

Вывод; в ходе выполненной работы, изучил и сделал конспект главы №3 “Processes” книги “Operating system concepts” by Abraham Silberschatz.