ТЕМА 2.1. ПРОЦЕССЫ ОПЕРАЦИОННЫХ СИСТЕМ

В данной теме рассматриваются следующие вопросы:

- Понятие процесса.
- Системные и пользовательские процессы.
- Операции над процессами.
- Контекст процесса.
- Динамически связываемые модули.
- Стандартные потоки ввода-вывода.
- Организация межпроцессного взаимодействия в ОС.
- Сигналы.
- Каналы.
- Классические проблемы межпроцессного взаимодействия.
- Файлы, отображаемые в память.
- Разделяемая память.

Лекции – 2 часа, лабораторные занятия – 2 часа, самостоятельная работа – 2 часа.

Экзаменационные вопросы по теме:

- Понятие процесса. Системные и пользовательские процессы. Операции над процессами.
- Организация межпроцессного взаимодействия в ОС. Сигналы. Каналы. Классические проблемы межпроцессного взаимодействия.

2.1.1. Понятие процесса

Процессы — это одна из самых старых и наиболее важных абстракций, присущих операционной системе. Они поддерживают возможность осуществления (псевдо) параллельных операций даже при наличии всего одного центрального процессора. Они превращают один центральный процессор в несколько виртуальных. Без абстракции процессов современные вычисления просто не могут существовать [1].

В любой многозадачной системе центральный процессор быстро переключается между процессами, предоставляя каждому из них десятки или сотни миллисекунд. При этом хотя в каждый конкретный момент времени центральный процессор работает только с одним процессом, в течение 1 секунды он может успеть поработать с несколькими из них, создавая иллюзию параллельной работы. Иногда в этом случае говорят о псевдопараллелизме в отличие от настоящего аппаратного параллелизма в многопроцессорных системах (у которых имеется не менее двух центральных процессоров, использующих одну и ту же физическую память). Людям довольно трудно отслеживать несколько действий, происходящих параллельно. Поэтому разработчики операционных систем за прошедшие годы создали **концептуальную модель последовательных процессов**, упрощающую работу с параллельными вычислениями.

В этой модели все выполняемое на компьютере программное обеспечение, иногда включая операционную систему, сведено к ряду последовательных процессов, или, для краткости, просто процессов. **Процесс** — это просто экземпляр выполняемой программы, включая текущие значения счетчика команд, регистров и переменных. Концептуально у каждого процесса есть свой, виртуальный, центральный процессор. Разумеется, на самом деле настоящий центральный процессор постоянно переключается между процессами, но, чтобы понять систему, куда проще думать о наборе процессов, запущенных в (псевдо) параллельном режиме, чем пытаться отслеживать, как центральный процессор переключается между программами. Это постоянное переключение между процессами называется **мультипрограммированием**, или **многозадачным** режимом работы.

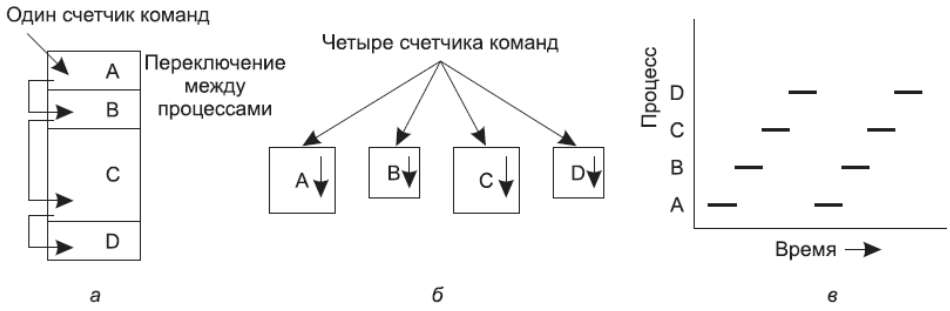

Рис. 2.1.1. Компьютер: а — четыре программы, работающие в многозадачном режиме; б — концептуальная модель четырех независимых друг от друга последовательных процессов; в — в отдельно взятый момент активна только одна программа

Поскольку центральный процессор переключается между процессами, скорость, с которой процесс выполняет свои вычисления, не будет одинаковой и, скорее всего, не сможет быть вновь показана, если тот же процесс будет запущен еще раз. Поэтому процессы не должны программироваться с использованием каких-либо жестко заданных предположений относительно времени их выполнения.

Разница между процессом и программой довольно тонкая, но весьма существенная. Ключевая идея здесь в том, что процесс — это своего рода действия. У него есть программа, входные и выходные данные и состояние. Один процессор может совместно использоваться несколькими процессами в соответствии с неким алгоритмом планирования, который используется для определения того, когда остановить один

процесс и обслужить другой. В отличие от процесса программа может быть сохранена на диске и вообще ничего не делать. Стоит отметить, что если программа запущена дважды, то считается, что ею заняты два процесса.

2.1.2. Системные и пользовательские процессы

Как говорилось в предыдущем пункте, при выполнении процесса операционная система назначает ему некоторый процессор. Процесс выполняет свои инструкции в течение некоторого периода времени. Затем он выгружается, освобождая процессор для другого процесса. Планировщик операционной системы переключается с кода одного процесса на код другого, предоставляя каждому процессу шанс выполнить свои инструкции.

Различают пользовательские процессы и системные [2].

Процессы, которые выполняют системный код, называются **системными** и применяются к системе в целом. Они занимаются выполнением таких служебных задач, как распределение памяти, обмен страницами между внутренним и вспомогательным запоминающими устройствами, контроль устройств и т.п. Они также выполняют некоторые задачи «по поручению» пользовательских процессов, например, делают запросы на ввод-вывод данных, выделяют память и т.д.

Пользовательские процессы выполняют собственный код и иногда обращаются к системным функциям. Выполняя собственный код, пользовательский процесс пребывает в пользовательском режиме (user mode). В пользовательском режиме процесс не может выполнять определенные привилегированные машинные команды. При вызове системных функций (например, **`read()`**, **`write()`** или **`open()`**) пользовательский процесс выполняет инструкции операционной системы. При этом пользовательский процесс «удерживает» процессор до тех пор, пока не будет выполнен системный вызов. Для выполнения системного вызова процессор обращается к ядру операционной системы. В это время о пользовательском процессе говорят, что он пребывает в привилегированном режиме, или режиме ядра (kernel mode), и не может быть выгружен никаким другим пользовательским процессом.

Фоновые процессы, предназначенные для обработки какой-либо активной деятельности, называются демонами в Unix и службами в Windows.

2.1.3. Операции над процессами

Итак, процесс — программа, которая выполняется в текущий момент.

Понятно, что реально на однопроцессорной компьютерной системе в каждый момент времени может исполняться только один процесс. Для мультипрограммных вычислительных систем псевдопараллельная обработка нескольких процессов достигается с помощью переключения процессора с одного процесса на другой. Пока один процесс выполняется, остальные ждут своей очереди. Каждый процесс может находиться как минимум в двух состояниях: *процесс исполняется* и *процесс не исполняется* (рис. 2.1.2) [3].

Рис. 2.1.2. Простейшая диаграмма состояний процесса

Процесс, находящийся в состоянии *процесс исполняется*, через некоторое время может быть завершен операционной системой или приостановлен и снова переведен в состояние процесс не исполняется [4].

Процесс, находящийся в состоянии «*процесс исполняется*», может быть либо завершен операционной системой, либо приостановлен и переведен в состояние «*процесс не исполняется*». Приостановка процесса происходит по двум причинам:

- для его дальнейшей работы требуется какое-либо событие (например, завершение операции ввода-вывода);
- истек временной интервал, отведенный операционной системой для работы данного процесса.

После приостановки процесса операционная система выбирает следующий процесс для исполнения, который находится в состоянии «процесс не исполняется» и переводит его в состояние «процесс исполняется» Новый процесс, появляющийся в системе, первоначально помещается в состояние «процесс не исполняется». Данная модель не отражает ситуацию ожидания выбранным процессом некоторого события. В процессе ожидания процесс к исполнению не готов. На рис. 2.1.3 изображена более подробная диаграмма состояний процесса.

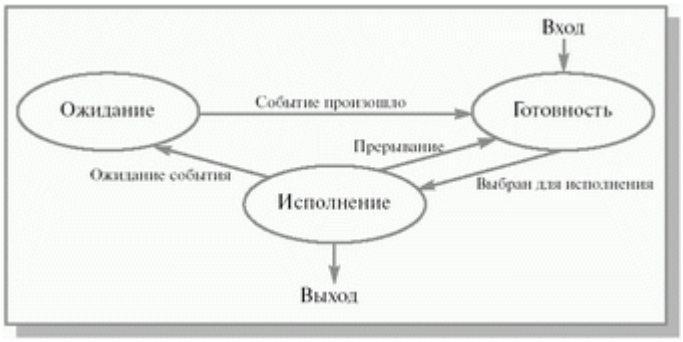

Рис. 2.1.3. Диаграмма состояний процесса с учетом состояний «ожидание» и «готовность»

В данной модели состояние «процесс не исполняется» разбивается на два состояния «готовность» и «ожидание». При появлении новый процесс попадает в состояние «готовность» и становится в очередь процессов, находящихся в состоянии «готовность». Операционная система в соответствии с результатами планирования выбирает один из них и переводит в состояние «исполнение». В состояние «исполнение» выполняется программный код процесса. Выйти из этого состояния процесс может по трем причинам:

- прекращение деятельности процесса;
- продолжение работы процесса зависит от наступления некоторого события, до наступления которого процесс переводится в состояние «ожидание», а при совершении события из состояния «ожидание» в состояние «готовность»;
- в результате прерывания операционная система возвращает процесс в состояние «готовность» (например, прерывания от таймера).

Данная модель описывает поведение процессов во время их существования, но не рассматривает случаи появления и исчезновения процесса в системе. На рисунке 2.1.4 представлена диаграмма состояний с учетом состояний «ожидания», «готовности», «рождения», «исполнения» и «закончил исполнение».

Рис. 2.1.4. Диаграмма состояний процесса с учетом состояний ожидания, готовности, рождения, исполнение и закончил исполнение.

Появление процесса в вычислительной системе соответствует состоянию «рождение». В состоянии «рождение» процесс получает в свое распоряжение адресное пространство для загрузки программного кода процесса, стек, системные ресурсы; ему устанавливается начальное значение программного счетчика процесса и т. д. По окончании «рождения» процесс переводится в состояние «готовность». При завершении деятельности процесс из состояния «исполнение» переходит в состояние «закончил исполнение».

Изложенная модель является наиболее общей и является основой всех операционных систем.

Процесс не может перейти из одного состояния в другое самостоятельно. Изменением состояния процессов занимается ОС, совершая операции над ними. Удобно объединить их в три пары:

- **создание процесса** – **завершение процесса**;
- **приостановка процесса** – **запуск процесса**;
- **блокирование процесса** – **разблокирование**.

Еще одна операция, не имеющая парной: **изменение приоритета процесса**.

Операции создания и завершения процесса являются одноразовыми, так как применяются к процессу не более одного раза (некоторые системные процессы при работе вычислительной системы не завершаются никогда). Все остальные операции являются многоразовыми.

Одноразовые операции приводят к изменению количества процессов, находящихся под управлением операционной системы, и всегда связаны с выделением или освобождением определенных ресурсов. Многоразовые операции, напротив, не приводят к изменению количества процессов в операционной системе и не обязаны быть связанными с выделением или освобождением ресурсов.

Существуют четыре основных события, приводящих к созданию процессов.

1. Инициализация системы.
2. Выполнение работающим процессом системного вызова, предназначенного для создания процесса.
3. Запрос пользователя на создание нового процесса.
4. Инициация пакетного задания.

При запуске операционной системы создаются, как правило, несколько процессов. Некоторые из них представляют собой высокоприоритетные процессы, то есть процессы, взаимодействующие с пользователями и выполняющие для них определенную работу.

Остальные являются фоновыми процессами, не связанными с конкретными пользователями, но выполняющими ряд специфических функций.

Операции над процессами.

Запуск процесса. Из числа процессов, находящихся в состоянии готовность, операционная система выбирает один процесс для последующего исполнения.

Приостановка процесса. Работа процесса, находящегося в состоянии исполнение, приостанавливается в результате какого-либо прерывания.

Блокирование процесса. Процесс блокируется, когда он не может продолжать работу, не дождавшись возникновения какого-либо события в вычислительной системе.

Разблокирование процесса. После возникновения в системе какого-либо события операционной системе нужно точно определить, какое именно событие произошло

Переключение контекста

В действительности же деятельность мультипрограммной операционной системы состоит из цепочек операций, выполняемых над различными процессами, и сопровождается переключением процессора с одного процесса на другой.

Создание процесса в Windows

В Windows одним вызовом функции **CreateProcess** создается процесс, и в него загружается нужная программа. У этого вызова имеется 10 параметров, включая выполняемую программу, параметры командной строки для этой программы, различные параметры безопасности, биты, управляющие наследованием открытых файлов, информацию о приоритетах, спецификацию окна, создаваемого для процесса (если оно используется), и указатель на структуру, в которой вызывающей программе будет возвращена информация о только что созданном процессе. В дополнение к функции **CreateProcess** в Win32 имеется около 100 других функций для управления процессами и их синхронизации, а также выполнения всего, что с этим связано [1].

Функция **CreateProcessW** создает новый процесс и его основной поток [5]. Новый процесс выполняется в контексте безопасности вызывающего процесса.

Если вызывающий процесс олицетворяет другого пользователя, новый процесс использует маркер для вызывающего процесса, а не маркер олицетворения. Чтобы запустить новый процесс в контексте безопасности пользователя, представленного маркером олицетворения, используйте функцию **CreateProcessAsUser** или **CreateProcessWithLogonW**.

Ниже приведен синтаксис функции **CreateProcessW**.

```
BOOL CreateProcessW(
  [in, optional]      LPCWSTR               lpApplicationName,
  [in, out, optional] LPWSTR                lpCommandLine,
  [in, optional]      LPSECURITY_ATTRIBUTES lpProcessAttributes,
  [in, optional]      LPSECURITY_ATTRIBUTES lpThreadAttributes,
  [in]                BOOL                  bInheritHandles,
  [in]                DWORD                 dwCreationFlags,
  [in, optional]      LPVOID                lpEnvironment,
  [in, optional]      LPCWSTR               lpCurrentDirectory,
  [in]                LPSTARTUPINFOW        lpStartupInfo,
  [out]               LPPROCESS_INFORMATION lpProcessInformation
);
```

lpApplicationName

Имя модуля для выполнения. Этот модуль может быть приложением на основе Windows. Это может быть другой тип модуля (например, MS-DOS или OS/2), если соответствующая подсистема доступна на локальном компьютере.

В строке можно указать полный путь и имя файла модуля для выполнения или указать частичное имя. В случае частичного имени функция использует текущий диск и текущий каталог для завершения спецификации. Функция не будет использовать путь поиска. Этот параметр должен включать расширение имени файла; Не предполагается расширение по умолчанию.

Параметр **lpApplicationName** может иметь значение NULL. В этом случае имя модуля должно быть первым маркером с разделителями пробелами в строке lpCommandLine.

lpCommandLine

Командная строка для выполнения. Максимальная длина этой строки составляет 32 767 символов, включая завершающий символ NULL в Юникоде. Если **lpApplicationName** имеет значение NULL, часть имени модуля в **lpCommandLine** ограничена MAX_PATH символами.

dwCreationFlags

Флаги, управляющие классом приоритета и созданием процесса. Флаги — это всегда отдельные биты, поэтому объединять их надо с помощью операции «логическое ИЛИ».

Для целей курса достаточно значений по умолчанию флагов создания процесса.

Флаги приоритета следующие:

```
IDLE_PRIORITY_CLASS            0x00000040
BELOW_NORMAL_PRIORITY_CLASS    0x00004000
NORMAL_PRIORITY_CLASS          0x00000020
ABOVE_NORMAL_PRIORITY_CLASS    0x00008000
HIGH_PRIORITY_CLASS            0x00000080
REALTIME_PRIORITY_CLASS        0x00000100
```

IDLE_PRIORITY_CLASS. Процесс, потоки которого выполняются только в том случае, если система простаивает и вытесняется потоками любого процесса, выполняющегося в классе с более высоким приоритетом. Примером является заставка. Класс неактивного приоритета наследуется дочерними процессами.

NORMAL_PRIORITY_CLASS. Процесс без особых потребностей в планировании.

HIGH_PRIORITY_CLASS. Процесс, выполняющий критически важные по времени задачи, которые необходимо выполнить немедленно, чтобы он выполнялся правильно. Потоки процесса класса с высоким приоритетом вытесняют потоки обычных или неактивных процессов класса приоритета. Примером является список задач, который должен быстро реагировать на вызов пользователем независимо от нагрузки на операционную систему.

REALTIME_PRIORITY_CLASS. Процесс, имеющий наивысший возможный приоритет. Потоки процесса класса приоритета в режиме реального времени вытесняют потоки всех остальных процессов, включая процессы операционной системы, выполняющие важные задачи. Например, процесс в режиме реального времени, который выполняется более чем за очень короткий интервал, может привести к тому, что кэши диска не будут очищаться или мышь перестает отвечать на запросы.

Возвращаемое значение. Если функция выполняется успешно, возвращается ненулевое значение. Если функция выполняется неудачно, возвращается нулевое значение.

Большинство процессов завершаются по окончании своей работы. Добровольно завершить собственный процесс в Windows можно вызовом функции **ExitProcess**. Другие процессы могут также завершить его вызовом функции **TerminateProcess** при наличии необходимых разрешений. Третий случай завершения процесса — при возникновении ошибки.

Создание процесса в Unix

С технической точки зрения во всех этих случаях новый процесс создается за счет уже существующего процесса, который выполняет системный вызов, предназначенный для создания процесса. Этим процессом может быть работающий пользовательский процесс, системный процесс, вызванный событиями клавиатуры или мыши, или процесс управления пакетными заданиями. Данный процесс осуществляет системный вызов для создания нового процесса. Этот системный вызов предписывает операционной системе создать новый процесс и прямо или косвенно указывает, какую программу в нем запустить.

В UNIX существует только один системный вызов для создания нового процесса — **fork**. Этот вызов создает точную копию вызывающего процесса. После выполнения системного вызова fork два процесса, родительский и дочерний, имеют единый образ памяти, единые строки описания конфигурации и одни и те же открытые файлы.

И больше ничего. Обычно после этого дочерний процесс изменяет образ памяти и запускает новую программу, выполняя системный вызов **execve** или ему подобный. Например, когда пользователь набирает в оболочке команду **sort**, оболочка создает ответвляющийся дочерний процесс, в котором и выполняется команда **sort**. Смысл этого двухступенчатого процесса заключается в том, чтобы позволить дочернему процессу управлять его файловыми дескрипторами после разветвления, но перед выполнением **execve** с целью выполнения перенаправления стандартного ввода, стандартного вывода и стандартного вывода сообщений об ошибках.

Для своего завершения процесс может вызвать функцию **exit**.

Только один процесс в системе рождается особенным способом — init — он порождается непосредственно ядром [6]. Все остальные процессы появляются путём дублирования текущего процесса с помощью системного вызова **fork**(2). После выполнения **fork**(2) получаем два практически идентичных процесса за исключением следующих пунктов:

- **fork**(2) возвращает родителю PID ребёнка, ребёнку возвращается 0;
- У ребёнка меняется PPID (Parent Process Id) на PID родителя.

После выполнения **fork**(2) все ресурсы дочернего процесса — это копия ресурсов родителя. Копировать процесс со всеми выделенными страницами памяти — дело дорогое, поэтому в ядре Linux используется технология Copy-On-Write.

Все страницы памяти родителя помечаются как read-only и становятся доступны и родителю, и ребёнку. Как только один из процессов изменяет данные на определённой странице, эта страница не изменяется, а копируется и изменяется уже копия. Оригинал при этом «отвязывается» от данного процесса. Как только read-only оригинал остаётся «привязанным» к одному процессу, странице вновь назначается статус read-write.

В некоторых программах реализована логика, в которой родительский процесс создает дочерний для решения какой-либо задачи. Ребёнок в данном случае решает какую-то конкретную проблему, а родитель лишь делегирует своим детям задачи. Например, веб-сервер при входящем подключении создаёт ребёнка и передаёт обработку подключения ему.

Однако, если нужно запустить другую программу, то необходимо прибегнуть к системному вызову **execve**(2):

```
int execve(const char *filename, char *const argv[], char *const envp[]);
```

или библиотечным вызовам **execl**(3), **execlp**(3), **execle**(3), **execv**(3), **execvp**(3), **execvpe**(3):

```
int execl(const char *path, const char *arg, ... /* (char *) NULL */);
int execlp(const char *file, const char *arg, ... /* (char *) NULL */);
...
```

И так далее… (**l** обозначает list, все параметры передаются как arg1, arg2, ..., NULL; **v** обозначает vector, все параметры передаются в нуль-терминированном массиве; **p** обозначает path; **e** обозначает environ)

Процесс при завершении (как нормальном, так и в результате не обрабатываемого сигнала) освобождает все свои ресурсы и становится «зомби» — пустой записью в таблице процессов, хранящей статус завершения, предназначенный для чтения родительским процессом.

Зомби-процесс существует до тех пор, пока родительский процесс не прочитает его статус с помощью системного вызова **wait**(), в результате чего запись в таблице процессов будет освобождена.

При завершении процесса система уведомляет родительский процесс о завершении дочернего с помощью сигнала **SIGCHLD**, таким образом может быть удобно (но не обязательно) осуществлять вызов **wait**() в обработчике данного сигнала.

Игнорирование обработки завершения дочерних процессов не является правильным, но обычно не приводит к проблемам для короткоживущих программ, так как при завершении процесса все его потомки становятся потомками процесса init, который постоянно считывает статус своих потомков-зомби, очищая таблицу процессов. Именно для задействования этого механизма выполняется стандартная техника запуска демонов "double fork()": промежуточный родитель завершается, делая родителем своего потомка процесс init .

Для долгоживущих и часто создающих дочерние процессы программ, необходима корректная обработка контроля завершения дочерних программ, потому что накапливание необрабатываемых зомби приводит к «утечке ресурсов» в виде накопления записей в таблице процессов.

В linux, начиная с версии kernel 3.4, процесс имеет возможность объявить себя усыновителем сирот ("subreaper") вместо процесса init командой **prctl(PR_SET_CHILD_SUBREAPER)**.

2.1.4. Контекст процесса

Для управления процессами операционной системе необходима информация об объекте управления, т.е. о процессах [7]. Процесс характеризуется следующей информацией:

- программный счетчик процесса, т.е. адрес команды для исполнения процесса;
- содержимое регистров процессора;
- данные, необходимые для планирования использования процессора и управления памятью (приоритет процесса, размер и расположение адресного пространства и т. д.);
- учетные данные (идентификационный номер процесса, какой пользователь инициировал его работу, общее время использования процессора данным процессом и т. д.);
- сведения об устройствах ввода-вывода, связанных с процессом (например, какие устройства закреплены за процессом, таблица открытых файлов).

Состав и строение данных о процессе зависят от конкретной операционной системы. В рамках курса Операционные системы считается, что информация о процессах доступна операционной системе и хранится в одной структуре данных, которая называется блоком управления процессом PCB (Process Control Block). Блок управления процессом является моделью процесса для операционной системы. Любая операция, производимая операционной системой над процессом, вызывает изменения в PCB. Содержимое PCB между операциями остается постоянным.

Информация, хранимая в блоке управления процессом, делится на две части: регистровый контекст и системный контекст. **Регистровым контекстом** процесса

называется содержимое всех регистров общего назначения процессора (включая значение программного счетчика). Содержимое остальных регистров процессора называется **системным контекстом** процесса. Информации, получаемой с регистровых и системных контекстов, достаточно для управления работой процесса в операционной системе.

С точки зрения пользователя, наибольший интерес вызывает вычислительная деятельность процесса, последовательность преобразования данных и полученные результаты. **Пользовательским контекстом** называются данные, находящиеся в адресном пространстве процесса.

> Совокупность регистрового, системного и пользовательского контекстов процесса называется **контекстом процесса**. В любой момент времени процесс полностью характеризуется своим контекстом.

2.1.5. Динамически связываемые модули

DLL (Dynamic Link Library — «библиотека динамической компоновки», «динамически подключаемая библиотека») в операционных системах Microsoft Windows — динамическая библиотека, позволяющая многократное использование различными программными приложениями. Эти библиотеки обычно имеют расширение DLL, OCX (для библиотек содержащих ActiveX), или DRV (для ряда системных драйверов). Формат файлов для DLL такой же, как для EXE-файлов Windows, то есть Portable Executable (PE) для 32-битных и 64-битных приложений Windows и New Executable (NE) — для 16-битных. Так же, как EXE, DLL могут содержать секции кода, данных и ресурсов. В системах Unix аналогичные функции выполняют так называемые общие объекты (англ. shared objects).

Файлы данных с тем же форматом как у DLL, но отличающиеся расширением или содержащие только секцию ресурсов, могут быть названы ресурсными DLL. В качестве примера можно назвать библиотеки значков, иногда имеющие расширение ICL, и файлы шрифтов, имеющих расширение FON и FOT.

Библиотеки не предназначены для запуска напрямую. Загружаются в память процесса загрузчиком программ операционной системы либо при создании процесса, либо по запросу уже работающего процесса, то есть динамически [8].

В UNIX-системах библиотеки имеют расширение `so` (shared object), в Windows — расширение `dll` (dynamic link library).

Динамические библиотеки могут использоваться двумя способами:

- динамическая компоновка (dynamic linking)
- динамическая загрузка (dynamic loading)

При динамической загрузке программа сама загружает конкретную библиотеку, указывая путь к ней, затем находит в библиотеке нужную функцию по имени и вызывает её. Это частый паттерн использования в программах, которые поддерживают плагины.

Переадресация (relocation)

Код в DLL обычно используется всеми процессами, использующими эту DLL; то есть они занимают одно место в физической памяти и не занимают место в файле подкачки. Windows не использует позиционно-независимый код для своих DLL; вместо этого код перемещается по мере загрузки, фиксируя адреса всех точек входа в местах, свободных в пространстве памяти первого процесса, загружающего DLL. В старых версиях Windows, в которых все запущенные процессы занимали одно общее адресное пространство, для всех процессов всегда было достаточно одной копии кода DLL. Однако в более новых версиях Windows, которые используют отдельные адресные пространства для каждой программы, можно использовать одну и ту же перемещенную копию DLL в нескольких

программах только в том случае, если каждая программа имеет одни и те же виртуальные адреса, свободные для размещения кода DLL. Если некоторые программы (или их комбинация уже загруженных DLL) не имеют этих свободных адресов, то необходимо будет создать дополнительную физическую копию кода DLL, используя другой набор перемещенных точек входа. Если необходимо освободить физическую память, занятую участком кода, ее содержимое отбрасывается, а затем при необходимости перезагружается непосредственно из файла DLL.

С учетом того, что разные программы имеют различные размеры и различный набор подгружаемых динамических библиотек, то если разделяемая библиотека отображается в адресное пространство различных программ, она будет иметь различные адреса. Это в свою очередь означает, что все функции и переменные в библиотеке будут на различных местах. Если все обращения к адресам относительные («значение +1020 байта отсюда») нежели абсолютные («значение в 0x102218BF»), то это не проблема, однако так бывает не всегда. В таких случаях всем абсолютным адресам необходимо прибавить подходящий офсет — это и есть relocation.

Это практически всегда скрыто от C/C++ программиста — очень редко проблемы компоновки вызваны трудностями переадресации.

Таблица перемещений (relocation table) — это список указателей, созданный транслятором (компилятором или ассемблером) и хранимый в объектном или исполняемом файле. Каждая запись в таблице, или «fixup», является указателем на абсолютный адрес в объектном коде, который должен быть изменен, когда загрузчик перемещает программу так, чтобы она ссылалась на правильное местоположение. Fixup'ы предназначены для поддержки переноса программы в виде цельной единицы.

ASLR

ASLR (англ. address space layout randomization — «рандомизация размещения адресного пространства») — технология, применяемая в операционных системах, при использовании которой случайным образом изменяется расположение в адресном пространстве процесса важных структур данных, а именно образов исполняемого файла, подгружаемых библиотек, кучи и стека.

Технология ASLR создана для усложнения эксплуатации нескольких типов уязвимостей. Например, если при помощи переполнения буфера или другим методом атакующий получит возможность передать управление по произвольному адресу, ему нужно будет угадать, по какому именно адресу расположен стек, куча или другие структуры данных, в которые можно поместить шелл-код. Если не удастся угадать правильный адрес, приложение скорее всего аварийно завершится, тем самым лишив атакующего возможности повторной атаки и привлекая внимание системного администратора.

Библиотеки импорта

Как и статические библиотеки, библиотеки импорта для DLL имеют расширение файла `.lib`. Например, kernel32.dll, основная динамическая библиотека для базовых функций Windows, таких как создание файлов и управление памятью, связана через kernel32.lib. Обычный способ отличить библиотеку импорта от правильной статической библиотеки — это размер: библиотека импорта намного меньше, поскольку она содержит только символы, относящиеся к фактической DLL, которые должны обрабатываться во время компоновки. Тем не менее, оба файла являются файлами формата `ar` Unix.

Связывание с динамическими библиотеками обычно осуществляется путем связывания с библиотекой импорта при сборке или связывании для создания исполняемого файла. Созданный исполняемый файл затем содержит таблицу адресов импорта (IAT), по которой ссылаются на все вызовы функций DLL (каждая ссылочная функция DLL содержит собственную запись в IAT). Во время выполнения IAT заполняется

соответствующими адресами, которые указывают непосредственно на функцию в отдельно загруженной DLL.

API и ABI

API: Application Program Interface

Это набор публичных типов, переменных, функций, которые вы делаете видимыми из вашего приложения или библиотеки.

В C и C++ API обычно поставляется в виде заголовочного файла (h) вместе с библиотекой.

С API работают люди, когда пишут код.

ABI: Application Binary Interface

Детали реализации этого интерфейса. Определяет такие вещи, как

- Способ передачи параметров в функции (регистры, стек).
- Кто извлекает параметры из стека (вызывающий код или вызываемый, caller/callee).
- Как происходит возврат значений из функции.
- Как реализован механизм исключений.
- Декорирование имён в C++ (mangling).

ABI важно, когда приложение использует внешние библиотеки. Если при обновлении библиотеки ABI не меняется, то менять программу не надо. API может остаться тем же, но поменяется ABI. Две версии библиотеки, имеющие один ABI, называют binary compatible (бинарно совместимыми): старую версию библиотеки можно заменить на новую без проблем.

Иногда без изменений ABI не обойтись. Тогда приходится перекомпилировать зависящие программы. Если ABI библиотеки меняется, а API нет, то версии называют source compatible.

Разработчики библиотеки стараются поддерживать ABI стабильным. Новые функции и типы данных могут добавляться, но старые должны сохраняться.

Windows

Несмотря на то что общие принципы разделяемых библиотек примерно одинаковы как на платформах UNIX, так и на Windows, всё же есть несколько существенных различий.

Экспортируемые символы

Самое большое отличие заключается в том, что в библиотеках Windows символы не экспортируются автоматически. В UNIX все символы всех объектных файлов, которые были подлинкованы к разделяемой библиотеке, видны пользователю этой библиотеки. В Windows программист должен явно делать некоторые символы видимыми, т. е. экспортировать их.

Есть **три способа** экспортировать символ и Windows DLL (и все эти три способа можно перемешивать в одной и той же библиотеке).

Первый способ

В исходном коде объявить символ как __declspec(dllexport), примерно так:

```
__declspec(dllexport) int my_exported_function(int x, double y)
```

Второй способ

При выполнении команды компоновщика использовать опцию LINK.EXE export:symbol_to_export

```
LINK.EXE /dll /export:my_exported_function
```

Третий способ

Скормить компоновщику файл определения модуля (DEF) (используя опцию /DEF:def_file), включив в этот файл секцию EXPORT, которая содержит символы, подлежащие экспортированию.

```
EXPORTS
  my_exported_function
    my_other_exported_function
```

Как приходится иметь дело с C++, первая из этих опций становится самой простой, так как в этом случае компилятор берёт на себя обязательства позаботиться о декорировании имён.

.LIB и другие относящиеся к библиотеке файлы

Вторая трудность, связанной с библиотеками Windows: информация об экспортируемых символах, которые компоновщик должен связать с остальными символам, не содержится в самом DLL. Вместо этого данная информация содержится в соответствующем .LIB файле.

.LIB файл, ассоциированный с DLL, описывает какие (экспортируемые) символы находятся в DLL вместе с их расположением. Любой бинарник, который использует DLL, должен обращаться к .LIB файлу, чтобы связать символы корректно.

Чтобы сделать всё ещё более запутанным, расширение .LIB также используется для статических библиотек.

На самом деле существует целый ряд различных файлов, которые могут относиться каким-либо образом к библиотекам Windows. Наряду с .LIB файлом, а также (опциональным) .DEF файлом Вы можете увидеть все нижеперечисленные файлы, ассоциированные с вашей Windows-библиотекой.

Файлы на выходе компоновки

- **library.DLL**: собственно код библиотеки; этот файл нужен (во время исполнения) любому бинарнику, использующему библиотеку.
- **library.LIB**: файл «импортирования библиотеки», который описывает, где и какой символ находится в результирующей DLL. Этот файл генерируется, если только DLL экспортирует некоторые её символы. Если символы не экспортируются, то смысла в .LIB файле нет. Этот файл нужен во время компоновки.
- **library.EXP**: «Экспорт файл» компилируемой библиотеки, который нужен, если имеет место компоновка бинарников с циклической зависимостью.
- **library.ILK**: Если опция /INCREMENTAL была применена во время компоновки, которая активирует инкрементную компоновку, то этот файл содержит в себе статус инкрементной компоновки. Он нужен для будущих инкрементных компоновок с этой библиотекой.
- **library.PDB**: Если опция /DEBUG была применена во время компоновки, то этот файл является программной базой данных, содержащей отладочную информацию для библиотеки.
- **library.MAP**: Если опция /MAP была применена во время компоновки, то этот файл содержит описание внутреннего формата библиотеки.

Файлы на входе компоновки

- **library.LIB**: Файл «импорта библиотеки», которые описывает где и какие символы находятся в других DLL, которые нужны для компоновки.

- **library.LIB**: Статическая библиотека, которая содержит коллекцию объектов, необходимых при компоновке. Обратите внимание на неоднозначное использование расширения .LIB
- **library.DEF**: Файл «определений», который позволяет управлять различными деталями скомпонованной библиотеки, включая экспорт символов.
- **library.EXP**: Файл экспорта компонуемой библиотеки, который может сигнализировать, что предыдущее выполнение LIB.EXE уже создало файл .LIB для библиотеки. Имеет значение при компоновке бинарников с циклическими зависимостями.
- **library.ILK**: Файл состояния инкрементной компоновки; см. выше.
- **library.RES**: Файл ресурсов, который содержит информацию о различных GUI-виджетах, используемых исполняемым файлом. Эти ресурсы включаются в конечный бинарник.

Это является большим отличием по сравнению с UNIX, где почти вся информация, содержащаяся в этих всех дополнительных файлах, просто добавляется в саму библиотеку.

Импортируемые символы

Вместе с требованием к DLL явно объявлять экспортируемые символы, Windows также разрешает бинарникам, которые используют код библиотеки, явно объявлять символы, подлежащие импортированию. Это не является обязательным, но даёт некоторую оптимизацию по скорости, вызванную историческими свойствами 16-битной Windows.

В LIB-файле для фунции FunctionName генерирутеся "заглушка", которая выглядит как

```
jmp [__imp__FunctionName]
```

Здесь __imp__FunctionName является записью в таблице импортированных функций. То есть заглушка считывает адрес из таблицы импортированных адресов (IAT) и выполняет переход на тот код.

За счёт двухступенчатого процесса получается лишняя потеря производительности.

Если писать dllimport, компилятор будет генерировать код для непрямого вызова через таблицу IAT прямо по месту. Это уменьшает число индирекций и позволяет компилятору локально (в вызывающей функции) закешировать целевой адрес. Хотя при наличии LTO это уже не актуально.

Объявляем символ как __declspec(dllimport) в исходном коде примерно так:

```
__declspec(dllimport) int function_from_some_dll(int x, double y);
__declspec(dllimport) extern int global_var_from_some_dll;
```

При этом индивидуальное объявление функций или глобальных переменных в одном заголовочном файле является хорошим тоном программирования на C. Это приводит к некоторому ребусу: код в DLL, содержащий определение функции/переменной, должен экспортировать символ, но любой другой код, использующий DLL, должен импортировать символ.

Стандартный выход из этой ситуации — это использование макросов препроцессора.

```
#ifdef EXPORTING_XYZ_DLL_SYMS
#define XYZ_LINKAGE __declspec(dllexport)
#else
#define XYZ_LINKAGE __declspec(dllimport)
#endif

XYZ_LINKAGE int xyz_exported_function(int x);
XYZ_LINKAGE extern int xyz_exported_variable;
```

Файл с исходниками в DLL, который определяет функцию и переменную гарантирует, что переменная препроцессора `EXPORTING_XYZ_DLL_SYMS` определена (посредством `#define`) до включения соответствующего заголовочного файла и таким образом экспортирует символ. Любой другой код, который включает этот заголовочный файл, не определяет этот символ и таким образом импортирует его.

Преимущества и недостатки динамических библиотек

Преимущества

- **Обновления и исправления ошибок.** Если нужно обновить код, который вынесен в динамическую библиотеку, то достаточно обновить только библиотеку, и все программы, что её используют, получат новую версию, их не надо пересобирать.
- **Экономия памяти.** Если одну и ту же библиотеку использует несколько приложений, в оперативной памяти может храниться только один ее экземпляр, доступный этим приложениям. Пример — библиотека C/C++. Ею пользуются многие приложения. Если всех их скомпоновать со статически подключаемой версией этой библиотеки, то код таких функций, как **`sprintf`**, **`strcpy`**, **`malloc`** и др., будет многократно дублироваться в памяти. Но если они компонуются с динамической версией библиотеки C/C++, в памяти будет присутствовать лишь одна копия кода этих функций, что позволит гораздо эффективнее использовать оперативную память.
- **Экономия места на диске.** Аналогично.
- **Общие данные.** Библиотеки могут содержать такие ресурсы, как строки, значки и растровые изображения. Эти ресурсы доступны любым программам.
- **Расширение функциональности приложения.** Библиотеки можно загружать в адресное пространство процесса динамически, что позволяет приложению, определив, какие действия от него требуются, подгружать нужный код. Поэтому одна компания, создав какое-то приложение, может предусмотреть расширение его функциональности за счет библиотек от других компаний.
- **Возможность использования разных языков программирования.** У вас есть выбор, на каком языке писать ту или иную часть приложения. Так, пользовательский интерфейс — на одном, прикладную логику — на другом.
- **Реализация специфических возможностей.** Определенная функциональность доступна только при использовании динамических библиотек. Например, в Windows отдельные виды ловушек (устанавливаемых вызовом **`SetWindowsHookEx`** и **`SetWinEventHook`**) можно задействовать при том условии, что функция уведомления ловушки размещена в DLL. Кроме того, расширение функциональности оболочки Windows возможно лишь за счет создания COM-объектов, существование которых допустимо только в DLL. Если говорить о UNIX, то примером специфической для динамических библиотек является возможность `LD_PRELOAD`.
- **Ускорение процесса сборки.** Каждый раз при сборке анализируются изменения только в коде приложения, выполняется более лёгкая компоновка.
- **Разработка в большой команде.** Если в процессе разработки программного продукта отдельные его модули создаются разными группами, то при использовании динамических библиотек таким проектом управлять проще.

Недостатки

- **Разработка в большой команде.** Затрудняется внесение изменений в код, нужно постоянно думать об обратной совместимости.

- **Неудобства при сборке.** Например, нужно собрать код программы с каким-то флагом компилятора, при этом приходится следить, чтобы все нужные библиотеки были также пересобраны с флагом и подхватились при запуске.
- **Неудобство развёртывания.** Статически скомпонованная программа — один самодостаточный файл.
- **Dependency hell.** Многочисленные проблемы разного вида. Например, две версии одной библиотеки, половина программ требуют первую, другая половина программ — вторую.
- **Потери производительности.** Вызов функции из динамической библиотеки получается медленнее.

2.1.6. Стандартные потоки ввода-вывода

Стандартные потоки ввода-вывода в системах типа UNIX (и некоторых других) — потоки процесса, имеющие номер (дескриптор), зарезервированный для выполнения некоторых «стандартных» функций. Как правило (хотя и не обязательно), эти дескрипторы открыты уже в момент запуска задачи (исполняемого файла).

Стандартный ввод

Поток номер **0 (stdin)** зарезервирован для чтения команд пользователя или входных данных.

При интерактивном запуске программы по умолчанию нацелен на чтение со стандартного устройства ввода (клавиатуры). Командная оболочка UNIX (и оболочки других систем) позволяют изменять цель этого потока с помощью символа «<». Системные программы (демоны и т. п.), как правило, не пользуются этим потоком.

Стандартный вывод

Поток номер **1 (stdout)** зарезервирован для вывода данных, как правило (хотя и не обязательно) текстовых.

При интерактивном запуске программы по умолчанию нацелен на запись на устройство отображения (монитор). Командная оболочка UNIX (и оболочки других систем) позволяют перенаправить этот поток с помощью символа «**>**». Средства для выполнения программ в фоновом режиме (например, **nohup**) обычно переназначают этот поток в файл.

Стандартный вывод ошибок

Поток номер **2 (stderr)** зарезервирован для вывода диагностических и отладочных сообщений в текстовом виде.

Чаще всего цель этого потока совпадает с **stdout**, однако, в отличие от него, цель потока **stderr** не меняется при «**>**» и создании конвейеров («**|**»). То есть, отладочные сообщения процесса, вывод которого перенаправлен, всё равно попадут пользователю. Командная оболочка UNIX позволяет изменять цель этого потока с помощью конструкции «**2>**». Например, для подавления вывода этого потока нередко пишется «2>/dev/null».

POSIX-функция обработки ошибок **perror** используется в языках программирования Си и C++ для вывода сообщения об ошибке в **stderr** на основе номера последней ошибки, хранящейся в `errno`.

PowerShell поддерживает следующие потоки вывода [9].

Поток#	Описание	Представлено в	Командлет Write
1	Success поток	PowerShell 2.0	Write-Output
2	Error поток	PowerShell 2.0	Write-Error
3	Warning поток	PowerShell 2.0	Write-Warning
4	Verbose поток	PowerShell 2.0	Write-Verbose

5	Debug поток	PowerShell 2.0	Write-Debug
6	Information поток	PowerShell 5.0	Write-Information
Н/Д	Progress поток	PowerShell 2.0	Write-Progress

2.1.7. Организация межпроцессного взаимодействия в ОС

При выполнении параллельных процессов может возникать проблема, когда каждый процесс, обращающийся к разделяемым данным, исключает для всех других процессов возможность одновременного с ним обращения к этим данным - это называется **взаимоисключением** (**mutual exclusion**).

Ресурс, который допускает обслуживание только одного пользователя за один раз, называется критическим ресурсом. Если несколько процессов хотят пользоваться критическим ресурсом в режиме разделения времени, им следует синхронизировать свои действия таким образом, чтобы этот ресурс всегда находился в распоряжении не более чем одного их них.

Для организации коммуникации между одновременно работающими процессами применяются средства межпроцессного взаимодействия (Interprocess Communication - IPC).

Выделяются три уровня средств IPC:

- локальный;
- удаленный;
- высокоуровневый.

Средства локального уровня IPC привязаны к процессору и возможны только в пределах компьютера.

К этому виду IPC принадлежат практически все основные механизмы IPC UNIX, а именно, каналы, разделяемая память и очереди сообщений.

Коммуникационное пространство этих IPC поддерживаются только в пределах локальной системы. Из-за этих ограничений для них могут реализовываться более простые и более быстрые интерфейсы.

Удаленные IPC предоставляют механизмы, которые обеспечивают взаимодействие как в пределах одного процессора, так и между программами на различных процессорах, соединенных через сеть.

Сюда относятся удаленные вызовы процедур (Remote Procedure Calls - RPC), сокеты Unix, а также TLI (Transport Layer Interface - интерфейс транспортного уровня) фирмы Sun.

Под **высокоуровневыми IPC** обычно подразумеваются пакеты программного обеспечения, которые реализуют промежуточный слой между системной платформой и приложением.

Эти пакеты предназначены для переноса уже испытанных протоколов коммуникации приложения на более новую архитектуру.

Методы IPC и их применимость к разным ОС

Файл. Все операционные системы.

Сигнал. Большинство операционных систем; некоторые системы, как например, Windows, только реализуют сигналы в библиотеке запуска Си, но не обеспечивают их полноценной поддержки для использования методов IPC.

Сокет. Большинство операционных систем.

Канал. Все системы, соответствующие POSIX.

Именованный канал. Все системы, соответствующие POSIX.

Семафор. Все системы, соответствующие POSIX.

Разделяемая память. Все системы, соответствующие POSIX.

Обмен сообщениями (без разделения). Используется в парадигме MPI, Java RMI, CORBA и других.

Проецируемый в память файл. Все системы, соответствующие POSIX; несет риск появления состояния гонки в случае использования временного файла. Windows также поддерживает эту технологию, но использует API отличный от POSIX.

Очередь сообщений. Большинство операционных систем.

Почтовый ящик. Некоторые операционные системы.

2.1.8. Сигналы

Сигнал в операционных системах семейства Unix — асинхронное уведомление процесса о каком-либо событии, один из основных способов взаимодействия между процессами. Когда сигнал послан процессу, операционная система прерывает выполнение процесса, при этом, если процесс установил собственный обработчик сигнала, операционная система запускает этот обработчик, передав ему информацию о сигнале, если процесс не установил обработчик, то выполняется обработчик по умолчанию.

Сигналы могут возникать синхронно с ошибкой в приложении, например SIGFPE (ошибка вычислений с плавающей запятой) и SIGSEGV (ошибка адресации), но большинство сигналов является асинхронными.

Сигналы могут посылаться процессу, если система обнаруживает программное событие, например, когда пользователь дает команду прервать или остановить выполнение, или получен сигнал на завершение от другого процесса.

Сигналы могут прийти непосредственно от ядра ОС, когда возникает сбой аппаратных средств ЭВМ.

Система определяет набор сигналов, которые могут быть отправлены процессу. В Linux применяется около 30 различных сигналов. При этом каждый сигнал имеет целочисленное значение и приводит к строго определенным действиям.

Отдельные сигналы подразделяются на три класса:

- системные сигналы (ошибка аппаратуры, системная ошибка и т.д.);
- сигналы от устройств;
- сигналы, определенные пользователем.

Механизм передачи сигналов состоит из следующих частей:

- установление и обозначение сигналов в форме целочисленных значений;
- маркер в строке таблицы процессов для прибывших сигналов;
- таблица с адресами функций, которые определяют реакцию на прибывающие сигналы.

Как только сигнал приходит, он отмечается записью в таблице процессов. Если этот сигнал предназначен для процесса, то по таблице указателей функций в структуре описания процесса выясняется, как нужно реагировать на этот сигнал. При этом номер сигнала служит индексом таблицы.

Известно три варианта реакции на сигналы:

- вызов собственной функции обработки;
- игнорирование сигнала (не работает для SIGKILL);
- использование предварительно установленной функции обработки по умолчанию.

Сигналы посылаются:

- с терминала, нажатием специальных клавиш или комбинаций (например, нажатие Ctrl-C генерирует SIGINT, Ctrl-\ SIGQUIT, а Ctrl-Z SIGTSTP);
- ядром системы:

- при возникновении аппаратных исключений (недопустимых инструкций, нарушениях при обращении в память, системных сбоях и т. п.);
- ошибочных системных вызовах;
- для информирования о событиях ввода-вывода;

- одним процессом другому (или самому себе):
  - из С-кода с помощью системного вызова kill():
  - из shell, утилитой /bin/kill (она позволяет задавать сигнал как числом, так и символьным обозначением).

Чтобы реагировать на разные сигналы, необходимо знать концепции их обработки. Процесс должен организовать так называемый обработчик сигнала в случае его прихода. Для этого используется функция signal():

```
#include <signal.h>
void(*signal(int signr, void(*sighandler)(int)))(int);
```

Сигналы ОС Linux

Номер	Значение	Реакция программы по умолчанию
SIGABRT	Ненормальное завершение (abort())	Завершение
SIGALRM	Окончание кванта времени	Завершение
SIGBUS	Аппаратная ошибка	Завершение
SIGCHLD	Изменение состояния потомка	Игнорирование
SIGCONT	Продолжение прерванной программы	Продолжение/ игнорирование
SIGEMT	Аппаратная ошибка	Завершение
SIGFPE	Ошибка вычислений с плавающей запятой	Завершение
SIGILL	Неразрешенная аппаратная команда	Завершение
SIGINT	Прерывание с терминала	Завершение
SIGIO	Асинхронный ввод/вывод	Игнорирование
SIGKILL	Завершение программы	Завершение
SIGPIPE	Запись в канал без чтения	Завершение
SIGPWR	Сбой питания	Игнорирование
SIGQUIT	Прерывание с клавиатуры	Завершение
SIGSEGV	Ошибка адресации	Завершение
SIGSTOP	Остановка процесса	Остановка
SIGTTIN	Попытка чтения из фонового процесса	Остановка
SIGTTOU	Попытка записи в фоновый процесс	Остановка
SIGUSR1	Пользовательский сигнал	Завершение
SIGUSR2	Пользовательский сигнал	Завершение
SIGXCPU	Превышение лимита времени CPU	Завершение
SIGXFSZ	Превышение пространства памяти (4GB)	Завершение
SIGURG	Срочное событие	Игнорирование
SIGWINCH	Изменение размера окна	Игнорирование

2.1.9. Каналы

Канал (pipe) представляет собой средство связи стандартного вывода одного процесса со стандартным вводом другого. Каналы старейший из инструментов IPC, существующий приблизительно со времени появления самых ранних версий оперативной системы UNIX. Для реализации IPC возможно использование полудуплексных и/или именованных каналов (FIFO).

Для создания канала используется системный вызов **pipe**(). Его формат следующий:

```
#include <inistd.h>
int pipe (int fd[2]);
```

Аргументом функции является указатель на массив двух целых чисел, в котором функция возвращает два файловых дескриптора. Первый из них предназначен для чтения данных из канала, второй - для записи в него.

После создания канала, процесс может при помощи обычного системного вызова **write**() выводить данные в него, а затем вводить их, вызывая соответственно функцию **read**(). При выполнении вызова **fork**() дескрипторы канала наследуются процессом-"потомком". Таким образом, оба процесса получают возможность обмениваться данными.

Ограничением в данном случае является то, что канал должен работать лишь в одну сторону. Либо "родитель" должен писать, а "потомок" читать, либо наоборот. В первом случае для передачи данных от процесса-"родителя" к процессу-"потомку", первый должен закрыть fd[0], а второй fd[1]. Во втором случае выполняются противоположные действия. Если предполагается передавать данные в обе стороны, необходимо создать два канала.

2.1.10. Классические проблемы межпроцессного взаимодействия

Синхронный доступ. Если все процессы считывают данные из файла, то в большинстве случае проблем не возникает. Однако, при попытке одним из процессов изменить этот файл, могут возникнуть так называемые конкурентные ситуации (race conditions).

Дисциплина доступа. Если нужно, чтобы как можно большее количество пользователей могли записывать данные, организуется так называемая очередь (по правилу «один пишет, несколько читают»). Практически организуется две очереди: одна — для чтения, другая — для записи. Такую дисциплину доступа можно организовать с помощью **барьеров** (блокировок). При этом создается общий барьер для считывателей, так как несколько процессов могут одновременно считывать данные, а также отдельный барьер для процесса-писателя. Такая организация предотвращает взаимные помехи в работе.

Голодание процессов. Организация дисциплины доступа может привести к ситуации, когда процесс будет длительно ждать своей очереди для записи данных. Поэтому иногда нужно организовывать очереди с приоритетами.

Если нельзя точно определить, какой из процессов запрашивает или возвращает свои данные в нужный компьютер первым, используется так называемое взаимодействие по модели "клиент-сервер". При этом используются один или несколько клиентов и один сервер. Клиент посылает запрос серверу, а сервер отвечает на него. После этого клиент должен дождаться ответа сервера, чтобы продолжать дальнейшее взаимодействие. Такое поведение называется **управлением потоком**. При одновременном доступе здесь также могут использоваться очереди с приоритетами.

Классический **тупик** возникает, если процесс А получает доступ к файлу А и ждет освобождения файла В. Одновременно процесс В, получив доступ к файлу В, ждет освобождения файла А. Оба процесса теперь ждут освобождения ресурсов другого процесса и не освобождают при этом собственный файл.

В литературе по операционным системам можно встретить множество интересных проблем использования различных методов синхронизации, ставших предметом широких дискуссий и анализа. В данном разделе мы рассмотрим наиболее известные проблемы.

Задача обедающих философов

В 1965 году Дейкстра сформулировал, а затем решил проблему синхронизации, названную им **задачей обедающих философов**. С тех пор все изобретатели очередного примитива синхронизации считали своим долгом продемонстрировать его наилучшие качества, показав, насколько элегантно с его помощью решается задача обедающих философов. Суть задачи довольно проста. Пять философов сидят за круглым столом, и у каждого из них есть тарелка спагетти. Эти спагетти настолько скользкие, что есть их можно только двумя вилками. Между каждыми двумя тарелками лежит одна вилка. Внешний вид стола показан на рис. 2.1.5.

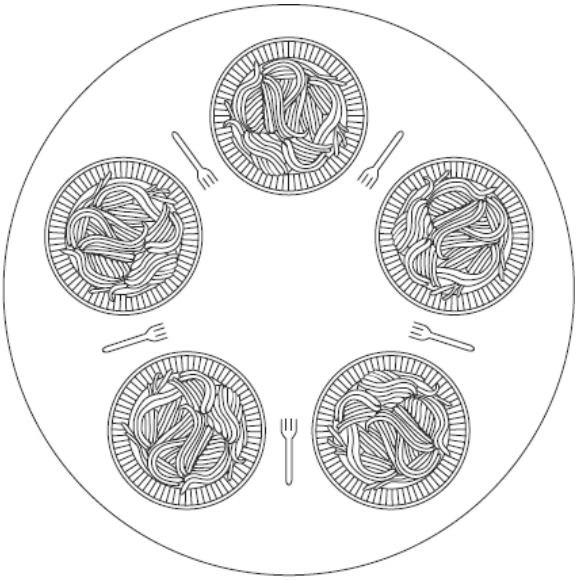

Рис. 2.1.5. Обед на факультете философии

Жизнь философа состоит из чередующихся периодов приема пищи и размышлений. Когда философ становится голоден, он старается поочередно в любом порядке завладеть правой и левой вилками. Если ему удастся взять две вилки, он на некоторое время приступает к еде, затем кладет обе вилки на стол и продолжает размышления. Основной вопрос состоит в следующем: можно ли написать программу для каждого философа, который действует предполагаемым образом и никогда при этом не попадает в состояние зависания?

Одно из решений использует массив состояний философов и семафор для защиты изменения состояний. В состояние EATING философ может перейти, только если в этом состоянии не находится ни один из его соседей.

Задача читателей и писателей

Другая общеизвестная задача касается читателей и писателей (Courtois et al., 1971). В ней моделируется доступ к базе данных. Представим, к примеру, систему бронирования авиабилетов, в которой есть множество соревнующихся процессов, желающих обратиться к ней для чтения и записи. Вполне допустимо наличие нескольких процессов, одновременно считывающих информацию из базы данных, но если один процесс обновляет базу данных (проводит операцию записи), никакой другой процесс не может получить доступ к базе данных даже для чтения информации. Вопрос в том, как создать программу для читателей и писателей?

В одном из решений первый читатель для получения доступа к базе данных выполняет в отношении семафора db операцию down. А все следующие читатели просто увеличивают значение счетчика rc. Как только читатели прекращают свою работу, они уменьшают значение счетчика, а последний из них выполняет в отношении семафора операцию up, позволяя заблокированному писателю, если таковой имеется, приступить к работе.

Этот алгоритм при наплыве читателей заблокирует писателя. Можно модифицировать алгоритм таким образом, что, при наличии в очереди писателя, новые читатели ставятся в очередь после писателя. Но это ведет к падению производительности системы.

2.1.11. Файлы, отображаемые в память

Отображение файла в память (на память) — это такой способ работы с файлами в некоторых операционных системах, при котором всему файлу или некоторой непрерывной части этого файла ставится в соответствие определенный участок памяти (диапазон адресов оперативной памяти). При этом чтение данных из этих адресов

фактически приводит к чтению данных из отображенного файла, а запись данных по этим адресам приводит к записи этих данных в файл.

Например, в Windows под "памятью" подразумевается не только оперативная память (ОЗУ), но также и память, резервируемая операционной системой на жестком диске. Этот вид памяти называется виртуальной памятью. Код и данные отображаются на жесткий диск посредством страничной системы (paging system) подкачки. Страничная система использует для отображения страничный файл pagefile.sys. Необходимый фрагмент виртуальной памяти переносится из страничного файла в ОЗУ и, таким образом, становится доступным.

Альтернативой отображению может служить прямое чтение файла или запись в файл. Такой способ работы менее удобен по следующим причинам:

- Необходимо постоянно помнить текущую позицию файла и вовремя ее передвигать на нужное место, в которое необходимо записать или из которого необходимо прочитать.
- Каждый вызов смены/чтения текущей позиции, записи/чтения — это системный вызов, который приводит к потере времени.
- Для работы через чтение/запись все равно приходится выделять буферы определенного размера, таким образом, в общем виде работа состоит из трех этапов: чтение в буфер -> модификация данных в буфере -> запись в файл. При отображении же работа состоит только из одного этапа: модификация данных в определенной области памяти.

Дополнительным преимуществом использования отображения является меньшая по сравнению с чтением/записью нагрузка на операционную систему.

При использовании отображений операционная система не загружает в память сразу весь файл, а делает это по мере необходимости, блоками размером со страницу памяти (как правило 4 килобайта). Таким образом, даже имея небольшое количество физической памяти (например, 32 мегабайта), можно легко отобразить файл размером 100 мегабайт или больше, и при этом для системы это не приведет к большим накладным расходам.

Также выигрыш происходит и при записи из памяти на диск: если вы обновили большое количество данных в памяти, они могут быть одновременно (за один проход головки над диском) сброшены на диск.

Наиболее общий случай, когда применяется отображение файлов на память — загрузка процесса в память (это справедливо и для Microsoft Windows и для Unix-подобных систем).

После запуска процесса операционная система отображает его файл на память, для которой разрешено выполнение (атрибут `executable`). Большинство систем, использующих отображение файлов, используют методику загрузка страницы по первому требованию, при которой файл загружается в память не целиком, а небольшими частями, размером со страницу памяти, при этом страница загружается только тогда, когда она действительно нужна.

В случае с исполняемыми файлами такая методика позволяет операционной системе держать в памяти только те части машинного кода, которые реально нужны для выполнения программы.

Чтобы воспользоваться этой возможностью в Unix-подобных системах, мы должны сообщить ядру о нашем желании отобразить файл в память. Делается это с помощью функции **mmap**() [10].

```
#include<sys/mman.h>
void *mmap(void *addr, size_t len, int prot, int flag, int filedes, off_t off);
```

Она возвращает адрес начала участка отображаемой памяти или MAP_FAILED в случае неудачи.

Первый аргумент — желаемый адрес начала участка отображённой памяти. Обычно не используется. Передаём 0 — тогда ядро само выберет этот адрес.

`len` — количество байт, которое нужно отобразить в память.

`prot` — число, определяющее степень защищённости отображенного участка памяти(только чтение, только запись, исполнение, область недоступна). Обычные значения — PROT_READ, PROT_WRITE (можно комбинировать через ИЛИ). Не буду на этом останавливаться — подробнее читайте в документации. Отмечу лишь, что защищённость памяти не установится ниже, чем права, с которыми открыт файл.

`flag` — описывает атрибуты области. Обычное значение — MAP_SHARED. По поводу остальных — подробнее читайте в документации. Но замечу, что использование MAP_FIXED понижает переносимость приложения, т.к. его подержка является необязательной в POSIX-системах.

`filedes` — дескриптор файла, который нужно отобразить.

`off` — смещение отображенного участка от начала файла.

В операционной системе Windows сначала файл открывается с помощью **`CreateFile`**(), и если открытие прошло успешно, то идентификатором файла можно воспользоваться для создания отображенного файла **`CreateFileMapping`**(), после которой мы получим идентификатор отображенного файла, на основе которого мы можем проецировать файл в память с помощью функции **`MapViewOfFile`**(). Эта функция даст нам указатель LPVOID, который используется для последующих операций, в том числе и по отмене проецированного файла **`UnmapViewOfFile`**() или принудительной записи на диск **`FlushViewOfFile`**(). В том случае, если с отображенным файлом будут работать несколько приложений одно приложение создает файл проецированный на память **`CreateFileMaping`**(), а остальные открывают эту проекцию **`OpenFileMapping`**().

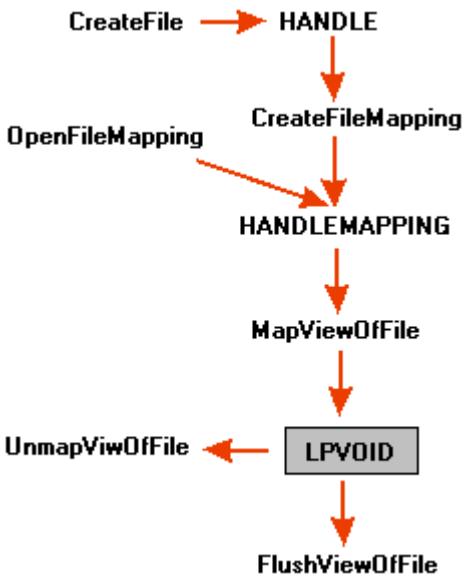

Рис. 2.1.6 Жизненный цикл отображения файла в память для Windows

2.1.12. Разделяемая память

В обеих системах, UNIX и Windows, после создания процесса родительский и дочерний процессы обладают своими собственными, отдельными адресными пространствами. Если какой-нибудь процесс изменяет слово в своем адресном пространстве, другим процессам эти изменения не видны. В UNIX первоначальное состояние адресного пространства дочернего процесса является копией адресного пространства родительского процесса,

но это абсолютно разные адресные пространства — у них нет общей памяти, доступной для записи данных. Некоторые реализации UNIX делят между процессами текст программы без возможности его модификации. Кроме того, дочерний процесс может совместно использовать всю память родительского процесса, но если память совместно используется в режиме копирования при записи (copy on write), это означает, что при каждой попытке любого из процессов модифицировать часть памяти эта часть сначала явным образом копируется, чтобы гарантировать модификацию только в закрытой области памяти. Следует также заметить, что память, используемая в режиме записи, совместному использованию не подлежит.

Тем не менее вновь созданный процесс может делить со своим создателем часть других ресурсов, например, открытые файлы. В Windows адресные пространства родительского и дочернего процессов различаются с самого начала.

В Linux для получения доступа к сегменту разделяемой памяти используется системный вызов **shmget**( ) [11]. Формат его следующий:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget (key_t key, int size, int flag);
```

Здесь:

`key` - уникальный ключ разделяемого сегмента памяти;

`size` - размер сегмента; в некоторых версиях системы UNIX он ограничен 128 К байтами;

`flag` - задание режима создания разделяемой памяти; значение этого параметра то же, что и в **msgget**( ), только префикс "MSG" заменен на "SHM".

При успешном создании или, если объект с заданным ключом key существует, функция возвращает идентификатор сегмента. В случае ошибки функция возвращает значение -1.

После создания разделяемой памяти, надо получить ее адрес. Для этого используется вызов **shmat**( ):

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
void *shmat (int shmid, void *addr, int flag);
```

Первый аргумент - идентификатор сегмента разделяемой памяти. Различные сочетания значений второго и третьего аргументов задают способ определения его адреса. Не приводя всех возможных вариантов, заметим, что простейшим вариантом из всех возможных является тот, при котором оба аргумента равны нулю. Возвращаемый системным вызовом **shmat**( ) адрес в дальнейшем используется процессом для прямого обращения к сегменту.

Изменить режим доступа к сегменту разделяемой памяти или удалить его можно при помощи системного вызова **shmctl**( ). При этом следует заметить, что сегмент не может быть удален до тех пор, пока не сделано обращение к процедуре **shmdt**( ). Прототипы **shmdt**( ) и **shmctl**( ) следующие:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
void shmdt (void *ptr);
int shmctl (int shmid, int cmd, struct shmid_ds *buf);
```

Здесь :

`ptr` - указатель на сегмент разделяемой памяти, полученный при вызове **shmat**( );

`shmid` - идентификатор сегмента, возвращаемый функцией **`shmget`**( );

`cmd` - выполняемая операция, основная из них IPC_RMID — удалить объект;

`buf` структура, используемая для передачи данных, необходимых для выполнения операции и/или получения ее результатов.

В операционной системе Windows первый процесс создает объект сопоставления файлов путем вызова функции **`CreateFileMapping`** с INVALID_HANDLE_VALUE и именем объекта [12]. С помощью флага PAGE_READWRITE процесс имеет разрешение на чтение и запись в памяти через все созданные представления файлов.

Затем процесс использует дескриптор объекта сопоставления файлов, возвращаемый **`CreateFileMapping`** в вызове **`MapViewOfFile`**, чтобы создать представление файла в адресном пространстве процесса. Функция **`MapViewOfFile`** возвращает указатель на представление файла, `pBuf`. Затем процесс использует функцию **`CopyMemory`** для записи строки в представление, к которому могут обращаться другие процессы.

Префикс имен объектов сопоставления файлов `"Global\"` позволяет процессам взаимодействовать друг с другом, даже если они находятся в разных сеансах сервера терминалов. Для этого требуется, чтобы первый процесс был иметь привилегию SeCreateGlobalPrivilege.

Если процессу больше не нужен доступ к объекту сопоставления файлов, он должен вызвать функцию **`CloseHandle`**. Когда все дескрипторы закрыты, система может освободить раздел файла подкачки, который использует объект.

Список использованных источников

1. Таненбаум, Э. Современные операционные системы. / Э. Таненбаум, Х. Бос. — 4-е изд. — СПб.: Питер, 2015. — 1120 с.

2. Параллельное и распределенное программирование на C++ *Хьюз Камерон*

3. Процессы и операции над ними
http://xn--80abdbnca6cjfb1beq.xn--p1ai/%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D1%8B-%D0%B8-%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%B8-%D0%BD%D0%B0%D0%B4-%D0%BD%D0%B8%D0%BC%D0%B8/

4. 2.2.Состояния процесса
https://studfile.net/preview/2014993/page:9/

5. CreateProcessW function (processthreadsapi.h)
https://learn.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createprocessw

6. Изучаем процессы в Linux
https://habr.com/ru/articles/423049/

7. Операции над процессами и связанные с ними понятия
https://studfile.net/preview/2014993/page:10/

8. C2017/Динамические библиотеки
https://acm.bsu.by/wiki/C2017/%D0%94%D0%B8%D0%BD%D0%B0%D0%BC%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B5_%D0%B1%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%D0%B8

9. О потоках вывода
https://learn.microsoft.com/ru-ru/powershell/module/microsoft.powershell.core/about/about_output_streams

10. Файлы, отображаемые в память
https://habr.com/ru/articles/55716/

11. Программирование в среде UNIX.
https://www.opennet.ru/docs/RUS/xtoolkit/x-1.html

12. Создание именованной общей памяти
https://learn.microsoft.com/ru-ru/windows/win32/memory/creating-named-shared-memory