## Тема 1.Разработка программ в ОС UNIX.

**Стандарты ОС UNIX**

Операционная система UNIX имеет долгую историю развития, существовало много ее официальных и фактических стандартов, а также коммерческих и учебных вариантов. Но ядро ОС UNIX остается стабильным, что в значительной степени определяет ее популярность и распространение.

В настоящее время стандарты UNIX определяются Posix и TheOpenGroup.

Posix (PortableOperatingSystemInterface) означает "интерфейс переносимых операционных систем". Это целое семейство стандартов, разработанное Институтом инженеров по электротехнике и радиоэлектронике (InstituteforElectricalandElektronicsEngineers (IEEE)).

Разработан ряд стандартов, определяющих правила применения системных вызовов и команд ОС UNIX.

POSIX 1003.1-1988 был первым стандартом Posix. Он определял интерфейс взаимодействия языка С c ядром UNIX в следующих областях: примитивы для реализации процессов, среда процесса, файлы и каталоги, работа с терминалом, базы данных систем (файлы паролей и групп), форматы архивов tar и cpio.

IEEE 1003.1-1990. Он одновременно являлся и международным стандартом ISO/IEC 9945-1:1990. Изменения по сравнению с первой версиеей были минимальными. к заголовку было добавлено "Part 1: SystemApplication Programming Interface (API)" (Часть 1: Системный интерфейс разработки программ”.

IEEE 1003.2-1992 определяет командный интерпретатор UNIX и набор утилит

IEEE 1003.1b-1993 включает дополнения, относящиеся к приложениям реального времени

IEEE 1003.1, издание 1996 года, включает 1003.1-1990 (базовый интерфейс API), 1003.1b (расширения реального времени), 1003.1c-1995 (Pthreads- программные потоки Posix) и 1003.1i-1995 (технические поправки к 1003.1b).

В 2001 году необязательные стандарты были объединены с базовым стандартом POSIX 1990, в результате появился стандарт IEEEStd 1003-2001. Последняя версия, выпущенная в 2004 году, носит название IEEEStd 1003-2004. Все ключевые стандарты POSIX обозначаются аббревиатурой POSIX.1, последняя версия датирована 2004 годом.

**Отличительные черты ОС UNIX**

1. Система написана на языке высокого уровня, что делает её доступной к пониманию, изменению и переносу на другие платформы. UNIX является одной из наиболее открытых систем.

2. UNIX – многозадачная, многопользовательская система. Один сервер может обслуживать запросы большого количества пользователей. При этом необходимо администрирование только одной системы.

3. Наличие стандартов. Основой семейства UNIX является принципиально одинаковая архитектура и ряд стандартных интерфейсов, что упрощает переход пользователей с одной системы на другую.

4. Простой, но мощный модульный пользовательский интерфейс. Имеется определенный набор утилит, каждая из которых решает узко специализированную задачу, и из них можно сконструировать сложные программные обрабатывающие комплексы.

5. Использование единой иерархической файловой системы, которая обеспечивает доступ к данным, хранящимся в файлах, и к устройствам системы.

6. Достаточно большое количество приложений, в том числе свободно распространяемых.

**Основы архитектуры операционной системы UNIX**

UNIX представляет собой двухуровневую модель системы: ядро и приложения. Ядро непосредственно взаимодействует с аппаратной частью компьютера, изолируя прикладные программы от аппаратных особенностей вычислительной системы. Ядро имеет набор услуг, предоставляемых прикладным программам.



Рис. 1.1. Модель системы

Все приложения запрашивают услуги ядра посредством системы вызовов.

Второй уровень составляют приложения или задачи, как системные, определяющие функциональность системы, так и прикладные, обеспечивающие пользовательский интерфейс UNIX. Схема взаимодействия всех приложений с ядром одинакова.

**Ядро системы**

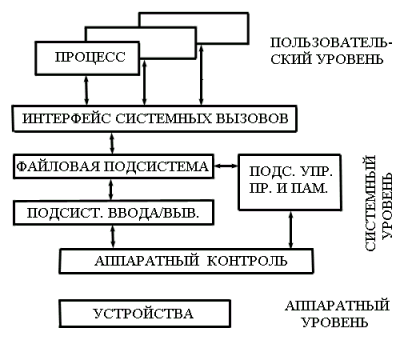


Рис 1.2. Ядро системы

**Ядро** обеспечивает базовую функциональность операционной системы, создает процессы и управляет ими, распределяет память и обеспечивает доступ к файлам и периферийным устройствам. Взаимодействие прикладных задач с ядром происходит посредством стандартного интерфейса системных вызовов.

Процесс запрашивает услугу определенной процедуры через стандартизированный системный вызов, внешне похожий на обычный вызов библиотечной функции. Ядро обрабатывает запрос и возвращает процессу необходимые данные.

Ядро состоит из основных трех подсистем:

- файловая подсистема;

- подсистема управления процессами и памятью;

- подсистема ввода-вывода.

**Файловая подсистема** обеспечивает унифицированный интерфейс доступа к данным, расположенным на дисковых накопителях и к периферийным устройствам. Одни и те же функции записи-чтения могут использоваться при работе с файлами на дисках и при вводе-выводе данных на терминал, принтер и другие внешние устройства.

Файловая подсистема контролирует права доступа к файлу, выполняет операции размещения и удаления файлов, запись и чтение данных.

Так как большинство прикладных функций использует в своей работе интерфейс файловой системы, права доступа к файлу во многом определяют привилегии доступа к системе. Таким образом формируются привилегии отдельных пользователей.

**Подсистема управления процессами и памятью.** Запущенная на выполнение программа порождает в системе один или более процессов. Подсистема управления процессами, контролирует создание и удаление процессов, распределение системных ресурсов между процессами, синхронизацию процессов, межпроцессорное взаимодействие.

Специальная задача ядра, называемая **планировщиком** процессов, разрешает конфликты между процессами в конкуренции за системные ресурсы. Планировщик запускает системные процессы и следит за тем, чтобы процесс не захватил разделяемые системные ресурсы.

Процесс освобождает процессор перед длительной операцией ввода-вывода или по прекращению кванта времени. В этом случае планировщик выбирает следующий процесс с наивысшим процессом и запускает его на выполнение.

Модуль управления памятью обеспечивает размещение оперативной памяти для прикладных задач. Если для всех процессов памяти недостаточно, ядро перемещает часть процесса или несколько процессов во вторичную память, представляющую собой специальную область на жестком диске, освобождающую ресурсы для выполняющегося процесса.

Все современные системы реализуют виртуальную память. При этом каждый процесс выполняется в своем логическом адресном пространстве, которое может значительно превышать физическую оперативную память. **Модуль управления памятью** выполняет функции виртуальной памяти.

**Модуль межпроцессорного взаимодействия** отвечает за уведомление процессов о событиях с помощью сигналов и обеспечивает возможность передачи данных между различными процессами.

**Подсистема ввода-вывода** обрабатывает запросы файловой подсистемы и подсистемы управления процессами для доступа к периферийным устройствам. Обеспечивает необходимость буферизации данных и взаимодействие с драйвером устройства.

**Драйверы**⎯ специальные системные программы, предназначенные для обеспечения взаимодействия процессов с устройствами.

**Пользователи системы, атрибуты пользователя**

Прежде чем клиент сможет начать работу с ОС UNIX, он должен стать пользователем системы, т.е. получить имя, пароль и ряд других атрибутов. С точки зрения системы пользователь - не обязательно человек. Пользователем является объект, который обладает определенными правами и может запускать на выполнение программы и владеть файлами. Пользователями могут быть отдельные люди, удаленные терминалы или группы пользователей с одинаковыми правами и функциями. В системе существует один пользователь, обладающий неограниченными правами - это суперпользователь или администратор системы.

Каждый пользователь имеет уникальное регистрационное имя, а система различает пользователей по идентификатору пользователя - UID. Идентификаторы также должны быть уникальны. Пользователи являются членами одной или нескольких групп. Группа - список пользователей, имеющих сходные задачи. Принадлежность к группе определяет дополнительные права, которыми обладают все пользователи группы. Каждая группа имеет уникальное имя, а система различает группы по групповому идентификатору (GID).

Идентификатор пользователя и идентификатор группы определяет, какими правами обладает пользователь в системе.

Информация о пользователях обычно хранится в специальном файле: /etc/passwd, о группах - /etc/group. Этот файл доступен только для чтения. Писать в него может только администратор. Каждая запись в файле содержит семь полей, разделенных: (двоеточием).

name: passwd-encode: UID: GID: comments: home-dir: shell

name - регистрационное имя пользователя (имя, которое вводится пользователем по приглашению **login**). В больших системах могут существовать определенные правила на выбор имени.

passwd-encode - пароль пользователя в закодированном виде. При входе в систему набираемый пароль кодируется и сравнивается с этим полем, и при совпадении пользователю разрешается вход в систему. Для повышения надежности системы часто пароли хранятся в отдельном файле, а это поле заполняется каким-то символом “x”. Пользователь, для которого в этом поле стоит “\*” не может попасть в систему, т.к. алгоритм кодирования не позволяет сформировать такой символ.

UID - идентификатор пользователя ⎯ внутреннее представление пользователя в системе. Наследуется задачами, которые запускает пользователь и файлами, которые создает пользователь. По этому идентификатору система проверяет права. Супер-пользователь имеет идентификатор 0.

GID - идентификатор первичной группы пользователя. Он соответствует идентификатору в файле etc/group, который содержит имя группы и полный список пользователей, являющихся её членами.

comments - может содержать расширенную информацию о пользователе (адрес, телефон и т.д.).

home-dir - домашний каталог пользователя. При входе в систему пользователь оказывается в этом каталоге. Как правило, пользователь имеет ограничения в других частях файловой системы. Но для домашнего каталога и подкаталогов он является полноправным хозяином.

shell - имя программы, которую ОС использует в качестве командного интерпретатора. Их существует несколько видов. Командный интерпретатор позволяет пользователю вводить команды и запускать задачи.

После запуска ОС UNIX в ней создаются несколько зарегистрированных пользователей:

root - суперпользователь с UID, равным 0. Пользователь с этим именем имеет неограниченные полномочия в системе. Для него не проверяются права доступа.

adm - псевдопользователь, владеющий файлами системы ведения журналов.

bin - владелец всех выполняемых файлов, являющихся командами UNIX.

cron - псевдопользователь, владеющий соответствующими файлами от имени которого выполняются процессы подсистемы запуска программ по расписанию.

lp или lpd - псевдопользователь, от имени которого выполняются процессы системы печати, и владеющий соответствующими файлами.

**Системные вызовы и функции стандартных библиотек**

Все версии UNIX предоставляют строго определенный ограниченный набор входов в ядро ОС, через которые прикладные задачи имеют возможность воспользоваться услугами, предоставляемыми ОС UNIX. Эти точки входа называются системными вызовами. Системный вызов определяет функцию, выполняемую ядром ОС от имени процесса, выполнившего вызов и является интерфейсом самого низкого уровня взаимодействия прикладных процессов с ядром. Современные ОС UNIX имеют более 100 системных вызовов. В среде программирования ОС UNIX системные вызовы определяются как функции языка C, независимо от фактической реализации вызова функции ядра ОС. В UNIX каждый системный вызов имеет соответствующую функцию или функции, с тем же именем, хранящуюся в стандартной библиотеке языка C.

Функции библиотеки выполняют необходимые преобразования элементов и вызывают требуемую процедуру ядра, используя различные приемы. В этом случае библиотечный код выполняет роль оболочки, а фактические инструкции располагаются в ядре операционной системы.

Файлы заголовков

Исходные тексты модулей программы

параметры

cc

1. компиляции

Объектные модули

библиотеки

ld

Выполняемый файл

a.out

Рис. 1.3. Схема компиляции программы

Программистам также предоставляется большой набор функций общего назначения, которые не являются точками входа в операционную систему, хотя в процессе выполнения многие из них выполняют системные вызовы (например, функция **printf** записывает данные в файл, используя системный вызов read). Библиотечные функции, хранящиеся в стандартных библиотеках вместе с системными вызовами, составляют основу среды программирования UNIX.

При разработке программ важное значение имеют файлы заголовков, где описаны значения большого количества констант и т.д.

**Описание программы, переменные окружения**

Выполнение программы начинается с создания в памяти её образа и связывания с процессом структур ядра ОС, инициализации и передачи управления инструкциям программы. Завершение программы приводит к освобождению памяти и соответствующих структур ядра. Образ программы в памяти содержит сегменты инструкций и данных, созданных компилятором, а также стек, которым пользуется программа в ходе выполнения.

Описание головной функции программы может выглядеть или так:

*main (int argc, char \*argv[], char \*envp[]);*

или так:

*extern char \*\*environ; // глобальная переменная, указывающая на строки*

*// с переменными окружения.*

*main (int argc, char \*argv[]);*

*argc* - определяет число параметров, переданных программе, включая её имя. Указатели на каждый из этих параметров хранятся в массиве, являющемся указателем на эти аргументы.

Второй массив - массив указателей на переменные окружения, передаваемые программе. Каждая переменная содержит строку вида:

HOME=/home/student/851003.

Основные переменные окружения следующие.

HOME - имя каталога, который становится текущим после входа пользователя в систему.

PATH - последовательность полных файловых путей, разделенных двоеточием, которые используются некоторыми программами для поиска файлов, заданных относительным именем.

TERM - тип терминала

TZ - информация о часовом поясе.

LOGNAME - регистрационное имя пользователя.

LC\_NUMERIC - значение категории локализации, определяющей правила национального представления чисел с плавающей точкой.

LC\_TIME - значение категории локализации, определяющей правила национального представления времени и даты.

Пример.

*#include <stddef.h>*

*#include <stdio.h>*

*#include <stdlib.h>*

*extern char \*\*environ;*

*void main(int argc, char \*argv[])*

*{*

*char \*ch;*

*char bufer[200], var[200];*

*if ((ch=getenv("MYVAR"))==NULL)*

*{*

*printf("Переменная MYVAR не определена. Введитезначение: ");*

*gets(bufer);*

*sprintf(var,"MYVAR=%s",bufer);*

*putenv(var);*

*printf("Новоезначение %s \n",var);*

*}*

*else*

*{*

*printf("MYVAR=%s. Изменить? ",ch);*

*gets(buf);*

*if (buf[0]=='Y' || buf[0]=='y')*

*{*

*printf("Новое значение: ");*

*gets(buf);*

*sprintf(var,"MYVAR=%s",bufer);*

*putenv(var);*

*printf("Новоезначение MYVAR=%s",var);*

*}*

*}*

*}*

*char \*getenv (const char \*name);*

*int putenv(const char \*string);*

Для получения и установки значений переменной окружения применяются две функции: getenv возвращает значение переменной окружения, имеющей имя name; putenv помещает переменную и ее значение в окружение программы. Переменные окружения позволяют передать программе некоторую информацию, однако большую часть информации программа получает от пользователя в ходе ее выполнения.

Вводимое новое значение переменной окружения будет действительно только для данного процесса и порожденных им процессов. После завершения текущего процесса измененная переменная окружения принимает исходное значение.

**Запуск и завершение программы**

При запуске программы на выполнение из командной строки shell автоматически устанавливает для нее три стандартных потока I/O (для ввода данных, вывода данных и для вывода сообщений об ошибках). Начальную связь их файловых дескрипторов с конкретными устройствами выполняет терминальный сервер. В большинстве случаев это процесс getty, который открывает специальный файл устройства, связанный с терминалом пользователя и получает соответствующие дескрипторы. Эти потоки наследует интерпретатор shell и передает их запускаемой программе. По умолчанию все три потока связаны с терминалом пользователя, но можено изменить стандартное направление потоков (перенаправить потоки) с помощью специальных команд.

При компиляции и создании программы редактор устанавливает точку входа в программу на специальную библиотечную функцию **\_start( )**. Эта функция инициализирует процесс, создавая кадр стека, устанавливая значение переменных, и затем вызывает функцию **main( )**.

Существует несколько способов завершения программы. Основным является возврат из функции **main( )** или вызов функции **exit( )**. Процесс также может завершиться по независящим от него обстоятельствам, например, при получении сигнала. Действие по умолчанию для большинства сигналов приводит к завершению процесса.

Прототип функции **еxit( )** записывается таким образом:

*#include <unistd.h>*

*void exit(int status);*

Аргумент **status**, передаваемый функции **exit( )**, передается родительскому процессу и представляет собой код возврата программы. По соглашению в случае успешного завершения программа возвращает 0, и другую величину (чаще всего -1) в противном случае.

Функция **exit( )** в ходе выполнения совершает ряд действий:

- выводит буферизованные данные;

- закрывает потоки ввода-вывода;

- при нормальном завершении процесса может вызывать специальные обработчики, которые предварительно могут быть установлены с помощью специальной функции **atexit( ).**

*#include<stdlib.h>*

*int atexit (void (\*func)(void));*

С помощью этой функции может быть зарегистрировано до тридцати двух обработчиков, и все они выполняются в порядке, обратном индексации.

Существует также функция **\_exit**, являющаяся системным вызовом.

**Обработка ошибок**

Обработке ошибок уделяется большое значение в UNIX, так как написание надежных и устойчивых программ, особенно для многопользовательских систем, является весьма важным.

Обычно в случае возникновения ошибки системные вызовы возвращают -1 и устанавливают значение переменной **errno**, указывающее возникновение ошибки. Библиотечные функции, как правило, значение **errno** не устанавливают, а код возврата различен для различных функций. Переменная **errno** не обнуляется следующим нормально завершившимся системным вызовом, следовательно эту переменную следует анализировать сразу же после системного вызова, который завершился с ошибкой.

Имеются также две функции, помогающие сообщить причину ошибочной ситуации.

*external int errno;*

*#include <string.h>*

*char \*strerror (int enum);*

*#include <errno.h>*

*#include <stdio.h>*

*void perror (char \*s);*

*main (int argc, char \*argv[ ])*

*{*

*fprintf(stderr, “ENOMEM:%s\n”, strerror (ENOMEM));*

*errno = ENOEXEC;*

*perror (argv[0]);*

*}*

Функция **strerror** принимает в качестве аргумента номер ошибки и возвращает указатель на строку, содержащую сообщение о причине ошибочной ситуации.

Функция **perror** выводит в стандартный поток сообщений об ошибках информацию об ошибочной ситуации, основываясь на значении переменной **errno**. Строка **s**, передаваемая функции в качестве аргумента, предваряет сообщение об ошибке и может содержать дополнительную информацию, например название функции, в которой произошла ошибка.

Имеется условное обозначение большого числа ошибочных ситуаций, при которых они могут обнаруживаться и обрабатываться. Некоторые из них приведены ниже.

|  |  |  |
| --- | --- | --- |
| № | Ошибочная ситуация | Примечание |
| 1 | E2BIG | Размер списка аргументов, переданных системному вызову exec плюс размер экспортируемых переменных окружения превышает максимально допустимое число байт (ARG\_MAX). |
| 2 | EACCESS | Попытка доступа к файлу с недостаточными правами доступа для данного класса. |
| 3 | EAGAIN | Превышен предел использования некоторого ресурса, например переполнена таблица процессов. |
| 4 | EBADF | Попытка операции с файловым дескриптором, не адресующим никакой файл или попытка работы с файлом на недопустимую операцию (например, файл открыт на чтение, а попытка записи). |
| 5 | EBADFD | Файловый дескриптор не адресует открытый файл или попытка выполнения операции чтения с файловым дескриптором, полученным при открытии только на запись. |
| 6 | EBUSY | Попытка монтирования устройства (файловой системы), которое уже примонтировано; попытка размонтировать файловую систему, имеющую открытые файлы; попытка обращения к недоступным ресурсам. |
| 7 | ECHILD | Вызов функции wait процессом, не имеющим дочерних процессов или повторный вызов wait для процесса, для которого этот вызов уже был сделан. |
| 8 | EDQUOT | Зарезервирован |
| 9 | EEXIST | Имя существующего файла использовано в недопустимом контексте, например, попытка создания символической связи с именем уже существующего файла. |
| 10 | EFAULT | Аппаратная ошибка при попытке использования системой аргумента функции, например, в качестве указателя передан недопустимый адрес. |
| 11 | EFBIG | Размер файла превысил установленное ограничение RLIMIT\_FSIZE или максимально допустимый размер для данной файловой системы. |
| 12 | EINTR | Получение асинхронного сигнала (SIGINT или SIGQUIT) во время обработки системного вызова.Если выполнение процесса будет продолжено после обработки сигнала, прерванный системный вызов завершится с этой ошибкой. |
| 13 | EINVAL | Передача неверного аргумента системному вызову. |
| 14 | EIO | Ошибка ввода-вывода физического устройства. |
| 15 | EISDIR | Попытка операции, недопустимой для каталогов. |
| 16 | EMFILE | Число открытых файлов для процесса превысило максимально допустимое значение. |
| 17 | ENAMETOOLONG | Длина имени файла, включая путь, превысило максимально допустимое значение. |
| 18 | ENFILE | Переполнение файловой таблицы. |
| 19 | ENODEV | Попытка недопустимой операции для устройства. |
| 20 | ENOENT | Файл с указанным именем не существует или отсутствует каталог, указанным в полном имени файла . |
| 21 | ENOEXEC | Попытка запуска на выполнение файла, который имеет права на выполнение, но не является файлом допустимого исполняемого формата |
| 22 | ENOMEM | Размер запрашиваемой памяти при запуске программы превысил максимально возможный в системе. |
| 23 | ENOTDIR | При задании имени каталога в операции, предусматривающей в качестве аргумента имя каталога, было указано имя файла другого типа. |
| 24 | EPIPE | Попытка записи в канал, для которого не существует процесса, принимающего данные. В этой ситуации процессу обычно направляется сигнал, ошибка возвращается при игнорировании сигнала. |
| 25 | ESRCH | Процесс с указанным PID не существует. |

**Заголовки в программах**

Использование системных функций требует включения в текст программы файлов заголовков, содержащих определение функций, типы аргументов и возвращаемые значения.

Файлы заголовков включаются в программу с помощью директивы **include**.

|  |  |  |
| --- | --- | --- |
| № | Библиотеки | Примечания |
| 1 | <assert.h> | Содержит прототип функции assert, используемой для диагностики |
| 2 | <cpio.h> | Содержит определения, используемые для файловых архивов cpio |
| 3 | <ctype.h> | Содержит определения символьных типов |
| 4 | <dirent.h> | Содержит определение структур данных каталога и прототипы функций работы с каталогами. |
| 5 | <errno.h> | Содержит определение кодов ошибок. |
| 6 | <float.h> | Содержит определение констант, необходимых для операций с плавающей точкой. |
| 7 | <ftw.h> | Содержит прототипы функций, используемых для сканирования дерева файловой системы. |
| 8 | <grp.h> | Содержит прототипы функций и определения структур данных, используемых для работы с группами пользователей |
| 9 | <longinfo.h> | Содержит определение языковых констант – дни недели и прототип функции longinfo. |
| 10 | <limits.h> | Содержит определение констант, определяющих максимальное и минимальное значение ограничений для данной реализации. |
| 11 | <locale.h> | Содержит определение констант, используемых для создания пользовательской среды, зависящей от языковых и культурных традиций |
| 12 | <math.h> | Содержит определение математических констант. |
| 13 | <nl\_types.h> | Содержит определения для каталогов сообщений |
| 14 | <pwd.h> | Содержит определение структуры файла паролей и прототипы функций работы с ним. |
| 15 | <regex.h> | Содержит определения констант и структур данных, используемых в регулярных выражениях |
| 16 | <search.h> | Содержит определения констант и функций, необходимые для поиска. |
| 17 | <setjmp.h> | Содержит прототипы функций перехода |
| 18 | <signal.h> | Содержит константы и функции для работы с сигналами. |
| 19 | <stdarg.h> | Содержит определения, необходимые для поддержки списков аргументов переменной длинны |
| 20 | <stddef.h> | Содержит стандартные определения, например типов. |
| 21 | <stdio.h> | Содержит определения стандартной библиотеки ввода-вывода |
| 22 | <stdlib.h> | Содержит определения стандартной библиотеки |
| 23 | <string.h> | Содержит прототипы функций для работы со строками |
| 24 | <tar.h> | Содержит определения, используемые для файловых архивов |
| 25 | <termios.h> | Содержит определения для обработки терминального вводы-вывода |
| 26 | <time.h> | Содержит определения для работы с датой и временем |
| 27 | <ulimit.h> | Содержит определения для управления ограничениями |
| 28 | <unistd.h> | Содержит определения системных символьных констант, а также прототипы большинства системных вызовов |
| 29 | <utime.h> | Содержит определения для работы с временными характеристиками файла |
| 30 | <sys/ipe.h> | Содержит определения, относящиеся к системе межпроцессорного взаимодействия |
| 31 | <sys/msg.h> | Содержит определения, относящиеся к сообщениям межпроцессорного взаимодействия |
| 32 | <sys/resourse.h> | Содержит определение констант и прототипы функций управления системными ресурсами. |
| 33 | <sys/sem.h> | Содержит определения, относящиеся к семафорам |
| 34 | <sys/shm.h> | Содержит определения, относящиеся к разделяемой памяти |
| 35 | <sys/stat.h> | Содержит определение структур данных и прототипы системных вызовов для получения информации о файле. |
| 36 | <sys/times.h> | Содержит определения для получения статистики выполнения процесса |
| 37 | <sys/types.h> | Содержит определение примитивов системных данных. |
| 38 | <sys/utsname.h> | Содержит определения для получения имен системы |
| 39 | <sys/wait.h> | Содержит определения используемые при синхронизации выполнения родственных процессов |
| 40 | <sys/fcntl.h> | Содержит определения, необходимые при работе с файлами |

**Стандартные форматы исполняемых файлов**

Виртуальная память процесса состоит из нескольких сегментов (областей) памяти. Размер, содержимое и расположение сегмента в памяти определяется как самой программой, так и форматом исполняемого файла.

Есть два стандартных формата исполняемых файлов.

COFF (Common Object File Format)

ELF (Executable and Linking Format)

Оба формата имеют сегмент кода (**text**), данных (**data**) и стека (**stack**). Размер сегментов **data** и **stack** может изменяться, а направление этого изменения определяется форматом исполнимого файла. Размер сегмента стека изменяется самой ОС, а управление размером сегмента **data** производится приложением с помощью специальных функций распределения памяти. Сегмент данных включает инициализированные данные, копируемые из специальных разделов исполнимого файла, и неинициализированные данные, которые заполняются 0 перед выполнением процесса. Неинициализированные данные часто называют сегментом BSS.

**Формат ELF**

|  |
| --- |
| Заголовок |
| Заголовок сегмента 1 |
| Заголовок сегмента 2 |
| … |
| Заголовок сегмента n |
| Сегмент 1 |
| Сегмент 2 |
| … |
| Сегмент n |
| Таблица заголовков секций |

Рис. 1.4. Структура исполняемого

файла в формате ELF

Существует три типа исполняемых ELF файлов:

1. Перемещаемый файл, содержащий инструкции и данные, которые могут быть связаны с другими объектными файлами. Результатом такого связывания может быть исполняемый файл или разделяемый объектный файл.

2. Разделяемый объектный файл. Также содержит инструкции и данные, но используется двумя способами. В первом случае он может быть связан с другими перемещаемыми файлами, в результате чего создается новый объектный файл. Во втором случае при запуске программы на выполнение операционная система может связать его динамически с исполняемым файлом программы. В результате чего создается исполняемый образ программы. Вообще речь идет о разделяемых библиотеках.

3. Исполняемые файлы. Хранит полное описание, то есть инструкции, данные, описание необходимых разделяемых объектных файлов, необходимую символьную и отладочную информацию, которая помогает операционной системе создать образ процесса.

Заголовок имеет фиксированное положение в файле, а остальные компоненты размещаются в соответствии с информацией, хранящейся в файле заголовка. То есть заголовок содержит общее описание структуры файла, расположение отдельных компонентов и их размеры.

|  |  |  |
| --- | --- | --- |
| № | Заголовки | Примечания |
| 1 | е\_ldent[ ] = ‘ ’,’ ’,’ ’… | Массив байт, каждый из которых определяет некую общую характеристику файла (формат файла, номер версии, архитектуру системы и т.д.). |
| 2 | e\_type = ET\_EXEC | Определяет тип файла. |
| 3 | e\_machine = EM388 | Архитектура аппаратной платформы, для которой создан данный файл. |
| 4 | e\_version = EV\_CURRENT | Номер версии ELF формата. Обычно ставится текущее значение EV\_CURRENC. |
| 5 | e\_entry | Виртуальный адрес, по которому системе будет передано управление после загрузки программы (то есть, точка входа). |
| 6 | e\_phoff | Расположение (то есть, смещение от начала файла) таблицы заголовков программы. |
| 7 | e\_shoff | Расположение таблицы заголовков секций. |
| 8 | e\_ehsize | Размер заголовка. |
| 9 | e\_phentsize | Размер каждого заголовка программы. |
| 10 | e\_phnum | Число заголовков программы. |
| 11 | e\_shentsize | Размер каждого заголовка сегмента или секции. |
| 12 | e\_shnum | Число заголовков сегментов или секций. |
| 13 | e\_shstrndx | Расположение сегмента, содержащего таблицу строк |

Информация, содержащаяся в таблице заголовком программы, указывает ядру операционной системы, как создать образ процесса из сегментов. Большинство сегментов копируются в память и представляют собой соответствующие сегменты процесса при его выполнении (например, сегмент кода или сегмент данных). Каждый заголовок сегмента программы описывает один сегмент и содержит следующую информацию:

1. Тип сегмента и действие операционной системы с этим сегментом.

2. Расположение сегмента в файле.

3. Стартовый адрес сегмента в виртуальной памяти процесса.

4. Размер сегментов в файле.

5. Размер сегментов в памяти.

6. Флаги доступа к сегменту.

Часть сегментов имеет тип LOAD, предписывающий ядру при запуске программы на выполнение создать соответствующие этим сегментам структуры данных, называемые областями, определяющие непрерывные участки виртуальной памяти процесса и связанные с ним атрибуты. К таким сегментам относятся сегменты, содержащие подпрограммы и ее данные.

В сегменте типа INTERР хранится программный интерпретатор. Данный тип сегмента используется для программ, которым необходимо динамическое связывание. Суть его состоит в том, что отдельные компоненты исполняемого файла подключаются не на этапе компиляции, а на этапе запуска программы на выполнение. Имя файла, являющегося динамическим редактором связей, хранится в данном сегменте. В начале в память загружается не исходная программа, а динамический редактор связей. Далее динамический редактор вместе с ядром UNIX создает полный образ исполняемого файла. То есть динамический редактор загружает необходимые разделяемые объектные файлы и производит требуемое размещение и связывание, а затем управление передается исходной программе.

Завершает файл таблица заголовков разделов или секций. Разделы определяют разделы файла, используемые для связи с другими модулями в процессе компиляции или при динамическом связывании. Разделы содержат более детальную информацию о сегментах. К примеру, сегмент кода может состоять из нескольких разделов, таких, как cash-таблица для хранения индексов, используемых в программе символов, раздел инициализационного кода программы, таблица связывания, используемая динамическим редактором и раздел, содержащий инструкции программы.

**Формат COFF**

|  |
| --- |
| Заголовок COFF |
| Заголовок a.out |
| Заголовок раздела 1 |
| Заголовок раздела 2 |
| … |
| Заголовок раздела n |
| Раздел 1 |
| Раздел 2 |
| … |
| Раздел n |
| Таблица символов |
| Таблица строк |

Рис. 1.3. Структура исполняемого

файла в формате COFF

Заголовок содержит общую информацию, позволяющую определить месторасположение остальных компонентов. Заголовок COFF содержит следующие поля:

|  |  |  |
| --- | --- | --- |
| № | Поля заголовка | Примечания |
| 1 | f\_magic | Аппаратная платформа, для которой создан файл. |
| 2 | f\_nscns | Количество разделов в файле. |
| 3 | f\_timdat | Время и дата создания файла. |
| 4 | f\_symptr | Расположение таблицы символов в файле. |
| 5 | f\_nsyms | Количество записей в таблице символов. |
| 6 | f\_opthdr | Размер заголовка a.out. |
| 7 | f\_flags | Флаги, указывающие на тип файлов, наличие символьной информации, таблицы строк и т.д. |

Заголовок COFF присутствует в исполняемых файлах, промежуточных объектных файлах и в библиотечных архивах. Каждый исполняемый файл также содержит заголовок **a.out**, хранящий информацию, необходимую ядру операционной системы или загрузчику для запуска программы.

|  |  |  |
| --- | --- | --- |
| № | Содержимое поля | Примечания |
| 1 | vstamp | Номер версии заголовка |
| 2 | tsize (text) | Размер раздела инструкций |
| 3 | dsize (data) | Размер раздела инициализированных данных |
| 4 | bsize (bss) | Размер раздела неинициализированных данных |
| 5 | Entry | Точка входа программы |
| 6 | text\_start | Адрес начала сегмента инструкций виртуальной памяти |
| 7 | data\_start | Адрес начала сегмента данных виртуальной памяти |

Все файлы формата COFF имеют два или более разделов, каждый из которых описывается своим заголовком. В заголовке хранится имя, размер раздела, его расположение в файле и виртуальный адрес после запуска программы на выполнение.

Таблица символов и строк является основой системы отладки.

Символом является любая переменная, имя функции или метка, определенные в программе. Каждая запись в таблице символов хранит имя символа, его виртуальный адрес, номер раздела, в котором определен символ, тип символа, класс хранения. Если имя символа занимает более восьми байт, то оно хранится в таблице строк. В этом случае в полном имени символа указывается смещение имени символа в таблице строк. Основное назначение символьной информации – применение в программах отладчиков, а также использование в программах, отражающих состояние процесса в системе.

***Тема 2. Файловая система ОС UNIX***

**Введение**

С точки зрения пользователей в UNIX есть два типа объектов: файлы и процессы. Все данные хранятся в виде файлов, а когда запускается программа, ядро загружает соответствующий исполняемый файл, создаёт образ процесса и передаёт ему управление. Во время работы процессы могут взаимодействовать с файлами.

Файлы в операционной системе играют одну из важнейших ролей, т.к. в них хранятся не только данные, но файлы также определяют и привилегии пользователей путём введения прав доступа к файлам. При обращении пользователя к файлу сравниваются права доступа, установленные для файла с правами конкретного пользователя. Файлы также обеспечивают доступ к периферийным устройствам машины.

Файлы организованы в древовидную файловую систему. Каждый файл имеет имя, определяющее его положение в дереве файловой системы. Корнем дерева является корневой каталог с именем “/”, имена всех остальных файлов содержат путь: **/home/student/proga.cpp**

Имя файла ⎯ атрибут файловой системы, а не набора некоторых данных на диске. Каждый файл имеет связанные с ним метаданные, которые хранятся в индексных дескрипторах. Индексный дескриптор содержит все характеристики файла, позволяющие операционной системе выполнять операции, затребованные прикладной задачей. В метаданных содержатся и указатели на дисковые области, в которых хранятся данные.

Имя файла в файловой системе ⎯ указатель на его метаданные. Сами метаданные не содержат указателя на имя файла. Полное имя файла не содержит указателя на физический носитель, где находится файл.

**Типы файлов**

В UNIX существуют следующие типы файлов:

1. Обычный файл
2. Каталог
3. Специальный файл устройства
4. Файл FIFO или именованный канал
5. Связь (link)
6. Сокет (socket)

**Обычный файл**: наиболее общий тип файлов, содержащий данные в некотором формате. Для операционной системы такие файлы представляют собой просто последовательность байтов. Интерпретация содержимого определяется прикладной программой, обрабатывающей файлы.

**Каталог**: файл, содержащий имена, находящихся в нём файлов, а также указатели на дополнительную информацию (индексные дескрипторы), позволяющую операционной системе производить операции над этими файлами. Каталоги определяют положение файла в дереве файловой системы, т.к. сам файл не содержит информации о своём местонахождении. По существу каталог представляет собой таблицу, где каждая запись соответствует одному файлу.

|  |  |
| --- | --- |
| Индексные дескрипторы | Имена файлов |
| 1753 | **.** |
| 2036 | **. .** |
| 751 | P1.txt |
| 854 | P2.c |

Рис. 2.1. Таблица каталога

**Специальный файл устройства**: обеспечивает доступ у физическому устройству. В UNIX различаются символьные и блочные файлы устройств. Символьные устройства применяются для не буферизированного обмена данными с устройствами (клавиатура, экран терминала), а блочные файлы ⎯ для обмена данными в виде пакетов фиксированной длины (дисковые накопители). К некоторым устройствам доступ может быть как блочный, так и символьный. Доступ к устройствам, как и к файлам, осуществляется путём открытия, чтения и записи в специальные файлы устройств.

**Файл FIFO** или именованный канал: специальный файл для связи между процессами - на запись и чтение и позволяет связать по данным несколько процессов. Поддержка файла FIFO появилась в UNIX, начиная с Release-5.

**Связь (link).** Индексные дескрипторы не содержать ни имени файла, ни указателя на него. Такой подход позволяет одному файлу иметь несколько имён в файловой системе. Имена жёстко связаны с индексными дескрипторами, и, соответственно, с данными файла, а сам файл существует независимо от того, как его называют, в файловой системе. Такая связь имени файла с его данными - жёсткая связь.

С помощью команды

**ln P1.dat /home/katalog2/Name2**

будет создана жёсткая связь.

Жёсткие связи абсолютно равноправны. Изменения, вносимые в файл, проявятся во всех файлах.

Жёсткая связь является естественной формой связи имени файла с его индексным дескриптором и не принадлежит к особому типу файлов.

**/home/katalog1 /home/katalog2**

|  |  |
| --- | --- |
| 534 | **.** |
| 734 | **. .** |
| 1374 | Name2 |
| 5374 | Sym\_name |

|  |  |
| --- | --- |
| 534 | **.** |
| 734 | **. .** |
| 1374 | P1.dat |

1374

Метаданные

5374

(inode)файла

Данные файла

Данные файла

/../katalog1/P1.dat

Рис. 2.2. Структура каталогов

Особым типом файла является символическая связь, позволяющая косвенно адресовать файл, который, в свою очередь, ссылается на другой файл. В результате последний файл в такой цепочке адресуется символической связью косвенно. В файле, который является символической связью, содержится только имя целевого файла. Создать символическую связь можно с помощью команды:

**ln -s Sym\_name /home/skatalog1/P1.dat**

Теперь по команде **cat Sym\_name** на экран будет выведено содержимое файла P1.dat.

Таким образом, при обращении к файлу, который является символической связью, действие переадресуется тому файлу, на который указывает связь.

**Сокет (socket)**: специальный файл, предназначенный для взаимодействия между процессами. Интерфейс сокетов часто используется для доступа к сети TCP/IP, в некоторых реализациях с его помощью осуществляется межпроцессорное взаимодействие.

/

home

Katalog2

Katalog1

Sym\_name

Name2

P1.dat

Жесткая связь

Символическая связь

Рис. 2.3. Каталоги с жесткой и символической связями

между файлами

**Структура файловой системы UNIX**

Корневой каталог ⎯ основа файловой системы UNIX. Все остальные файлы и каталоги располагаются в рамках структуры, порождённой корневым каталогом, независимо от их физического местонахождения.

В **/bin** находятся наиболее часто употребимые команды и утилиты системы.

В **/dev** находятся специальные файлы устройств, являющиеся интерфейсом доступа к периферийным устройствам. Этот каталог может содержать несколько подкаталогов, группирующих несколько файлов устройств одного типа.

В **/etc** находятся системные конфигурационные файлы и утилиты администрирования. Самые важные ⎯ скрипты инициализации системы, которые хранятся в каталогах rcN, где N ⎯ номер, определяющий уровень выполнения системы.

В **/default** находятся параметры, задающиеся по умолчанию для многих команд.

В **/lib** находятся библиотечные файлы C и других языков программирования.

Каталог **/lost+found**⎯ каталог потерянных файлов. При аппаратных сбоях и сбоях операционной системы могут появляться безымянные файлы. Программы проверки и восстановления помещают сюда неповреждённые безымянные файлы под числовыми именами.

/

home

mnt

bin

etc

tmp

usr

lost+found

rdsk

dsk

dev

lib

bin

lib

spool

include

local

cron

lp

male

default

rc2.d

rc3.d

init.d

rc0.d

Рис. 2.4. Структура файловой системы UNIX

Каталог **/mnt**⎯ стандартный каталог для временного связывания (монтирования) физических файловых систем к корневой для получения единого дерева логической файловой системы. Обычно содержимое этого каталога пусто, т.к. при монтировании он перекрывается связанной файловой системой.

Каталог **/home** предназначен для размещения каталогов пользователей.

Каталог **/usr**⎯ каталог различных сервисных систем.

В каталоге **/spool** находятся выполняемые файлы утилит UNIX.

Каталог **/tmp** предназначен для хранения временных файлов для работы UNIX.

**Владельцы файлов**

Файлы в UNIX имеют двух владельцев: пользователя и группу. **Группой** называется некий список пользователей системы. Пользователь может быть членом нескольких групп, одна из которых является первичной, другие ⎯ дополнительными. Владелец-пользователь может не быть членом группы, владеющей файлом. Это обеспечивает возможность организовать гибкую систему владения файлами для любого состава пользователей. Для определения владельцев файла используется команда **ls – l** для просмотра файлов каталогом. При этом в двух столбиках выводится имя владельца файла и имя владельца-группы.

Владельцем-пользователем вновь созданного файла является пользователь, создавший файл. Но реально файл создает не пользователь, а процесс, запущенный пользователем.

Процесс имеет атрибуты, связанные с пользователем и группой, которые назначаются в файл при его создании. Идентификатор владельца-пользователя устанавливается равным эффективному пользовательскому идентификатору процесса, создающего файл. Порядок назначения владельца-группы зависит от конкретной реализации операционной системы. Примером может быть назначение первичной группы владельца-пользователя. Владельцем-группой может стать группа, владеющая каталогом, в котором создан файл. Для изменения владельца файла используется команда **chown**:

chown stud1 p1.c p2.c t.txt

Первый параметр определяет пользователя, а далее следует список файлов, пользователем (владельцем) которых становится субъект, определяемый первым аргументом.

Для изменения владельца-группы используется команда:

chgrp ppt \*.c

Владение файлом определяется набором операций, которые пользователь может совершать с файлом. Изменение прав доступа или изменение владельца файла может осуществлять только владелец или суперпользователь. Операции чтения-записи или запуск на выполнение дополнительно контролируются правами доступа.

**Права доступа к файлу**

В операционной системе UNIX существует три базовых класса доступа к файлу, в каждом из которых установлены права доступа к файлу. Эти классы следующие.

1. Класс владельца-пользователя файла.

2. Класс членов группы, являющейся владельцем файлов.

3. Класс остальных пользователей, кроме суперпользователя.

Поддерживается три типа прав доступа для каждого класса:

- на чтение;

- на запись;

- на выполнение.

Если выполнить команду **ls –l**, то на экран выводится полный список файлов для каталога, который является текущим :

- | rwx rwx rwx 1.stud1 gr1 … f1.dat

- | r-x r-x - - - 2.stud2 gr1 … f2.dat

*тип файла категории пользователя*

Первый символ определяет тип файла:

d - каталог,

c - специальное символьное устройство,

p – поименованный FIFO файл,

- - обычный файл,

b - блоковое устройство.

Группы из трех символов:

I - права доступа пользователя,

II - права доступа группы,

III - права остальных пользователей.

Обозначения внутри группы:

- - нет доступа по данной категории,

r - право на чтение,

w - право на запись,

x - право на выполнение.

Права доступа могут быть изменены только владельцем файла или суперпользователем.

Для этого используется команда:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | u | + | r |  |
| chmod | g | - | w | f1 f2 f3 |
|  | o | = | x |  |
|  | a |  |  |  |

chmod a + w t.txt

chmod g+x-w r.out

Команда имеет следующие аргументы.

Первый символ - определение классов доступа:

u - владелец-пользователь,

g - владелец-группа,

o - остальные пользователи,

a - все классы пользователей.

Следующий символ - операция, которую следует провести:

+ - добавить права,

- - удалить права,

= - присвоить права.

Следующий символ определяет права, которые прибавляются, удаляются или присваиваются, а далее - список файлов, по отношению к которым выполняется действие.

Таким образом, можно изменять права доступа к файлам.

Значение прав доступа различно для разных типов файлов. Например, чтобы просмотреть содержимое файла с помощью команды **cat**, пользователь должен иметь право на чтение. Для редактирования файла и его изменения должно быть право на запись. Для запуска программы на выполнение должно быть право на выполнение. Например, для того, чтобы распечатать файл на принтере, необходимо право на запись специального устройства, связанного с принтером.

Права на доступ к каталогам имеют специфические особенности. К примеру, право чтения каталога позволяет получить лишь имена файлов, находящихся в данном каталоге. Для получения дополнительной информации, например получения команды **ls –l**, требуется обращение к индексным дескрипторам файла, что требует наличия прав выполнения для каталога. Право выполнения каталога требуется и для команды **cd**. Права чтения и выполнения для каталогов действуют независимо. Комбинацией этих прав можно добиться создания “темных каталогов”, файлы которых доступны только в том случае, если пользователь знает заранее их имена, так как получение списка файлов в таких каталогах запрещено. Этот подход может быть использован для построения некоторых справочных систем, когда отдельным пользователям сообщаются имена разделов, с которыми они могут работать, а остальные файлы им недоступны.

Создание и удаление файлов в каталоге требует изменения его содержимого и следовательно, права на запись в этот каталог. При этом не учитываются права доступа для самого файла, то есть для удаления файла из каталога не требуется каких-либо прав доступа к файлу, нужно иметь лишь право на доступ для каталога, в котором содержится файл.

Операционная система проводит проверку прав доступа при создании, открытии для чтения или записи, запуске на выполнение или удалении файлов. При этом выполняются следующие проверки:

1. Если операция запрашивается суперпользователем, никакие дополнительные проверки не производятся. Это позволяет администратору иметь неограниченный доступ ко всей файловой системе.

2. Если операция запрашивается владельцем файла, то, если требуемое право доступа определено, доступ разрешается. В противном случае - доступ запрещается.

3. Если операция запрашивается пользователем, являющимся членом группы, которая является владельцем файла, то, если требуемое право доступа определено, доступ разрешается, в противном случае ⎯ запрещается.

4. Если требуемое право доступа для прочих пользователей установлено, то доступ разрешается, в противном случае доступ запрещается.

Система проводит проверки в данной последовательности. К примеру, если пользователь является владельцем файла, то доступ определяется исключительно из прав доступа владельца. Права владельца группы не проверяются, даже если пользователь является членом владельца-группы.

**Дополнительные атрибуты файлов по управлению правами****доступа**

Существует несколько дополнительных атрибутов, изменяющих стандартное выполнение различных операций. Это относится как к обычным файлам, так и к каталогам.

|  |  |
| --- | --- |
| t | Sticky bit |
| s | Set UID, SUID |
| s | Set GID, SGID |
| l | Блокирование |

t - сохранить образ выполняемого файла после завершения;

s - установить пользовательский идентификатор процесса при выполнении;

s - установить групповой идентификатор процесса при выполнении;

1 - установить обязательное блокирование файлов при выполнении

Для установления атрибутов используется та же команда

chmod u+s f1

Атрибут **t**: после завершения выполнения задачи ее образ (код и данные) остаются в памяти, что приводит к тому, что последующие запуски программы занимают намного меньше времени.

Атрибуты **SUID** и **SGID** позволяют изменить права пользователя при запуске на выполнение файла, имеющего эти атрибуты. Обычно запускаемая программа, имеет права доступа к системным ресурсам на основе прав доступа пользователя, запустившего программу. Установка флагов SUID или SGID изменяет эти правила, назначая права доступа владельцев исполняемых файлов. То есть, если владельцем-пользователем является суперпользователь, то неограниченные права доступа к системным ресурсам получает и пользователь, запустивший этот файл.

Атрибут **1** используется для устранения конфликтов, когда одновременно несколько задач работают с одним и тем же файлом.

Для каталогов устанавливаются только два дополнительных параметра: t и s. Установка t для каталога позволяет установить дополнительную защиту файлов, находящихся в каталоге. Из такого каталога пользователь может удалить только те файлы, которыми он владеет или на которые он имеет право доступа на запись, даже при наличии прав на запись в каталог.

Это используется при работе с каталогом временных файлов ТМР, который открыт на запись для всех пользователей, но в котором нежелательно удаление каталогов файлов других пользователей.

Атрибут **SGID** тоже имеет специальное значение для каталогов. При установке этого атрибута для каталога вновь созданные файлы этого каталога будут наследовать владельца-группу по владельцу-группе каталога.

**Устройства**

ОС UNIX изолирует приложения от аппаратной части вычислительной системы. Она предоставляет единый интерфейс различных устройств системы в виде специальных файлов устройств. Специальный файл связывает прикладное приложение с драйвером устройства. Каждый специальный файл соответствует какому-либо физическому устройству (диск, устройство печати, терминал).

Вся работа приложения с устройством происходит через специальный файл, а соответствующий ему драйвер обеспечивает выполнение операций I/O в соответствии с конкретным протоколом обмена данными с устройством. Существуют файлы блочных устройств и файлы символьных устройств. Файлы блочных устройств обеспечивают интерфейс устройствам, обмен данными с которыми происходит большими фрагментами - блоками. При этом ядро ОС обеспечивает нужную буферизацию.

Файлы символьных устройств используются для доступа к устройствам, драйверы которых обеспечивают собственную буферизацию и побайтную передачу данных (накопители на магнитной ленте, терминалы). Одно и тоже устройство может иметь как блочный, так и символьный интерфейсы. Названия специальных файлов устройств зависят от конкретной версии UNIX, но при этом присутствуют некоторые правила названия таких устройств:

c*k*t*l*d*m*s*n*,

где k - номер контроллера;

l - номер устройства;

m - номер раздела;

n - логический номер устройства.

/dev/rmtn - накопитель на магнитной ленте;

/dev/cdn - CD-ROM;

/dev/ttypn - подчиненный псевдотерминал;

/dev/console - системная консоль;

/dev/tty - терминальная линия управляющего терминала для данного процесса;

/dev/mem - физическое ОЗУ;

/dev/kmem - виртуальная память ядра;

/dev/null - нулевое устройство, весь вывод на него пропадает, а при попытке ввода из этого устройства возвращается 0 байт информации;

/dev/zero - нулевое устройство, весь вывод на него уничтожается, а ввод приводит к получению последовательности нулей.

При выводе каталога **ls -l** можно определить, какое это специальное устройство и какой номер драйвера используется при его работе.

*Пример:*

*/\* вывод информации на принтер \*/*

*#include <fcntl.h>*

*main()*

*{ int I, fd;*

*fd=open(“/dev/lp”,O\_WRONLY);*

*for (i=0;i<500;i++) write(fd,”x”,1);*

*close(fd);*

*}*

*Прототип:*

*#include <sys/statvfs.h>*

*int statvfs(const char \*path, struct statvfs \*buf);*

*int fstatvfs(int fd,struct statvfs \*buf);*

*struct statvfs*

*{*

*unsigned long f\_bsize*; - размерблокаданных,

при которой наибольшая производительность

*unsigned long f\_frsize*; - основной размер блока в файловой системе

*unsigned long f\_block*; - полное число блоков

*unsigned long f\_bfree*; - полное число свободных блоков

*unsigned long f\_avail*; - число свободных блоков,

доступных непривелигированным пользователям

*unsigned long f\_files*; - полное число номеров индексных дескрипторов

*unsigned long f\_ffree*; - полное число свободных номеров ИД

*unsigned long f\_favail*; - число номеров ИД, доступных непривелигированным процессам

*unsigned long f\_fsid*; - идентификатор файловой системы

*unsigned long f\_flag*; - битовая маска значений флагов

*unsigned long f\_namemax*; - максимальная длина файла

*}*

**Системные вызовы и библиотечные функции**

В среде программирования UNIX существует два основных интерфейса для файлового ввода-вывода:

1. Системные вызовы, предлагающие основные функции работы с файлами, непосредственно взаимодействующие с ядром операционной системы.

2. Стандартная библиотека функций ввода-вывода.

К основным системным функциям для работы с файлами относятся:

*#include <sys/types.h>*

*#include <sys/stat.h>*

*#include <fcntl.h>*

*#include <unistd.h>*

*intopen(constchar \*pathname.tntflags [, mode\_tmode])* - предназначен для получения доступа на чтение (запись, чтение-запись) к указанному файлу. Если файл существует, он открывается и процессу возвращается файловый дескриптор, используемый при выполнении последующих действий с фалом (при выполнении операций чтения, записи и т.п.).

*int creat (const char \*pathname, mode\_t mode)* - применяетсядлясозданияфайла.

*int сlose* - закрывает файловый дескриптор.

*dup* - возвращает дубликат файлового дескриптора.

*dup2* - возвращает дубликат файлового дескриптора, но еще позволяет явно задать его значение.

*off\_t lseek (intfiledes, off\_toffset, intstont\_flag)* - устанавливает файловый указатель на определенное место файла. Последующие операции чтения (записи) будут производиться, начиная с этого смещения.

*ssize\_t read (intfiledes, void \*buffer,site\_tn)*- осуществляет чтение заданного количества байтов из файла.

*readv* - осуществляет несколько операций чтения заданного количества байтов из файла.

*ssize\_t write (intfiledes, void \*buffer,site\_tn)* - выполняет запись заданного количества байтов в файл.

*writev* - выполняет несколько операций записи заданного количества байтов в файл.

*pipe* - создает канал для передачи данных, возвращая два файловых дескриптора, один для выполнения операций чтения, а другой - операций записи.

*int fcntl (intfiledes, intcmd, …)* - обеспечивает выполнение функций управления открытым файлом.

*Примеры:*

*/\* копирование одного файла в другой \*/*

*#include <unistd.h>*

*#include <fcntl.h>*

*#define BUFSIZE 512*

*int copyfile(const char \*name1,const char \*name2)*

*{ int in\_f,out\_f;*

*ssize\_t n;*

*char buffer[BUFSIZE];*

*if ((in\_f=open(name1,O\_RDONLY))==-1) return (-1);*

*if ((out\_f=open(name2,O\_WRONLY|O\_CREAT|O\_TRUNC,0777)==-1)*

*{*

*close(in\_f);*

*return (-2);*

*}*

*while ((n=read(in\_f,buffer,BUFSIZE))>0)*

*{*

*if (write(out\_f,buffer,n)<n)*

*{*

*close(in\_f);*

*close(out\_f);*

*return (-3);*

*}*

*}*

*close(in\_f);*

*close(out\_f);*

*if (n==-1) return (-4);*

*else return (0);*

*}*

*/\* отображение содержимого файла порциями \*/*

*#include<sys/types.h>*

*#include <unistd.h>*

*#include <stdio.h>*

*#define NAMELENGTH 41*

*#define N\_ROOMS 20*

*char name\_buf[NAMELENGTH];*

*int in\_f=1;*

*char \*get\_occupior ( int room\_no)*

*{ off\_t offset;*

*size\_t nread;*

*if ((in\_f==-1) && (in\_f=open(“residents”,O\_RDONLY))==-1)*

*return(NULL);*

*offset=(room\_no-1)\*NAMELENGTH;*

*if (lseek(in\_f,offset,SEEK\_SET)==-1) return (NULL);*

*if (nread=read(in\_t,name\_buf,NAMELENGTH)) return(NULL);*

*namebuf[nread-1]=”\0”;*

*return(namebuf);*

*}*

*main()*

*{ int j;*

*char \*p;*

*for (j=1;j<=N\_ROOMS;j++)*

*{*

*if (p=get\_occupior(j)) printf(“Комната %d, %s\n”,j,p0;*

*else printf(“Ошибка”);*

*}*

*close(in\_f);*

*}*

*/\* отображениестатусафайла \*/*

*int file\_status(int filedes)*

*{ int arg;*

*if ((arg=fcntl(filedes,F\_GETFL))==-1)*

*{ printf(“Ошибка чтения статуса файла\n”) return (-1); }*

*switch(arg&ACCMODE)*

*{*

*case O\_WRONLY : printf(“Толькодлязаписи\n”); break;*

*caseO\_RDWR : printf(“Для чтения и записи\n”); break;*

*caseO\_RDONLY : printf(“Только для чтения”); break;*

*default : printf(“Режим не существует\n”);*

*}*

*if (arg&O\_APPEND) printf(“Установлен режим дозаписи\n”);*

*return(0);*

*}*

Порядок проверки прав доступа при обращениии к файлам

Если пользователь является владельцем файла, то проверяются те права, которые установлены для категории владельца файла. Если они разрешены – то действие выполняется, иначе - нет. Если действия выполняет пользователь группы владельцев, то проверяются права группы пользователей. Дополнительные права доступа для исполняемых файлов :

04000 S\_ISVID (1)

02000 S\_ISGOD (2)

01000 S\_ISVIX (3)

1. Права доступа проверяются по идентификатору владельца файла;
2. Для группы владельцев;
3. Бит сохранения сегмента кода (для каталогов);

Реальный идентификатор – идентификатор, который получается при входе в систему. Эффективный идентификатор получается при проверке прав доступа к файлу. В большинстве случаев реальный и эффективный иднтификаторы совпадают.

Проверка прав

доступа

запускает

на

выполнение

student

суперпользователь

a.exe

файлы

Маскасозданияфайла

*fildes=open(pathname,O\_CREAT, mode);*

*fildes=open(pathname,O\_CREAT,(~mask)&mode);*

*mode\_tumask(mode\_tnewmask) – системный вызов для установки маски*

С каждым процессом связана маска создания файла для автоматического выключения битов прав доступа при создании файла не зависимо от режима заданных опций функций open или creat. Это предотвращает случайное включениел лишних прав доступа при создании выполняемым процессом файлов.

*Прототип:*

*#include <sys/types.h>*

*#include <sys/stat.h>*

*mode\_t umask ( mode\_t newmask);*

*Пример:*

*/\* функция позволяет создать файл с правами,*

*значения которых передаются через второй аргумент \*/*

*#include <fcntl.h>*

*#include <sys/stat.h>*

*int special\_creat(const char \*pathname,mode\_t mode)*

*{ mode\_t oldu;*

*int f\_d;*

*if (oldu=umask(0))==-1)*

*{ printf(“Ошибкасохранениястароймаски\n”); return (-1); };*

*if ((f\_d=open(pathname,O\_WRONLY|O\_CREAT|O\_EXCL,mode))==-1)*

*printf(“Ошибка открытия файла\n”);*

*if (umask(oldu)==-1) printf(“Ошибка восстановления старой маски\n”);*

*return(f\_d);*

*}*

*Прототип:*

*#include <unistd.h>*

*int acces(const char \*pathname, int amode);*

*Пример:*

*/\* проверка отдельных прав доступа к файлу \*/*

*#include <stdio.h>*

*#include <stdlib.h>*

*#include <unistd.h>*

*main()*

*{*

*char \*filename=”afile”;*

*if (access(filename,R\_OK)==0)*

*printf(“Пользователь имеет право доступа на чтение\n”);*

*if (access(filename,W\_OK)==-1)*

*printf(“Пользователь не имеет права доступа на запись\n”);*

*if (access(filename,X\_OK)==0)*

*printf(“Есть право выполнения\n”);*

*}*

Стандартная библиотека ввода-вывода вместо использования файлов дескриптора определяет указатель на специальную структуру данных типа FILE, называемый потоком или файловым указателем. Стандартные потоки ввода-вывода обозначаются символическими именами: stdin, stdout и stderr соответственно для потоков ввода, вывода и сообщения об ошибках. Они соответствуют файловым дескрипторам 0, 1 и 2 и определены следующим образом:

*extern FILE \*stdin;*

*extern FILE \*stdout;*

*extern FILE \*stderr.*

Для работы с файлами используются следующие наиболее часто используемые функции:

fopen - открывает файл с указанным именем и возвращает файловый указатель, ассоциированный с данным файлом.

fclose - закрывает поток, освобождая буферы.

fflush - очищает буфер потока, открытого на запись.

getc - считывает символ из потока.

putc - записывает символ в поток.

gets - считывает строку из потока.

puts - записывает строку в поток.

fread - считывает указанное число байтов из потока.

fwrite - записывает указанное число байтов в поток.

fseek - позиционирует указатель в потоке.

fprintf - производит форматированный вывод.

fscanf - производит форматированный ввод.

fileno - возвращает файловый дескриптор данного потока.

Основным достоинством библиотечных функций является буферизация ввода/вывода, позволяющая минимизировать число системных вызовов чтения и записи. При открытии файла и создании потока библиотечные функции автоматически размещают необходимые буферы, о которых приложение может уже не заботиться.

Обеспечивается три типа буферизации:

1. Полная буферизация. Операции чтения и записи завершаются после того, как будет заполнен буфер ввода/вывода. Содержимое буфера записывается на диск автоматически или при вызове функции **fflush**.

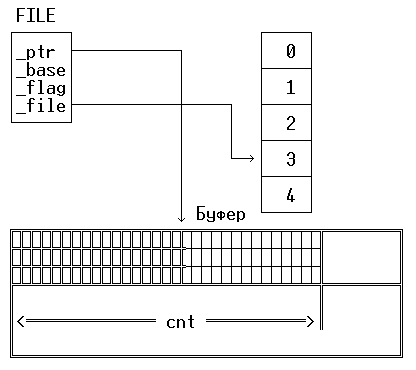
2. Построчная буферизация. Практический ввод/вывод выполняется с помощью системных вызовов **read**, **write** построчно при обнаружении конца строки (перевода каретки). Обычно применяется для стандартных потоков ввода/вывода.

3. Отсутствие буферизации. Функции библиотеки не производят никакой буферизации, являясь лишь программной оболочкой системных вызовов. При этом достигаются минимальные задержки операции чтения или записи. Применяется для стандартного потока вывода сообщений об ошибках.

Характер буферизации может быть изменен с помощью следующих функций:

*void setbuf (FILE \*stream , char \*buf);*

*int setbuf (file \*stream, char \*buf, int type, size\_t size);*



BUFSIZE или stat.st\_blksize

Рис. 2.5. Структура данных потока

Функция **setbuf** позволяет включить/выключить буферизацию для потока, указанного в качестве первого аргумента. Если второй аргумент указывает на буфер размером **szie**, то буферизация включается, а если тут стоит NULL, то выключается.

Функция **setvbuf** позволяет установить тип буферизации (третий аргумент). Её четвёртый аргумент определяет размер буфера, адресованного вторым аргументом type, который может принимать следующие значения:

\_IOFBF - полная буферизация;

\_IOLBF - построчная буферизация;

\_IONBF - отсутствие буферизации.

Каждый поток в стандартной библиотеке представлен указателем на структуру FILE, которая содержит следующие поля:

1. Указатель на следующий символ, подлежащий записи (\_ptr).

2. Число байт в буфере (\_cnt).

3. Указатель на буфер (\_base).

4. Флаги состояния потока (\_flag).

5. 4Указатель на файловый дескриптор (\_file).

Размер буфера определяется либо значением специального поля st\_blksize в структуре **stat**, возвращаемой системным вызовом stat, либо с помощью стандартного значения BUFERSIZE, определённого в библиотеке <stdio.h>.

**Файлы отображаемые в памяти**

*#include <sys/types.h>*

*#include <sys/mman.h>*

*caddr\_t mmap(caddr\_t addr, size\_t len, int prot, int flags, int fields, off\_t off);*

|  |  |
| --- | --- |
| prot | flags |
| PROT\_READ | MAP\_SHARED |
| PROT\_WRITE | MAP\_PRIVATE |
| PROT\_EXEC | MAP\_FIXED |
| PROT\_NONE | MAP\_NORESERVE |

Системный вызов **mmap** предоставляет доступ к файлам, альтернативный вызовам **read** и **write**. С помощью этого вызова процесс может отображать участки файлов в своё адресное пространство. После этого данные файла могут быть получены или записаны путем чтения, записи в память.

Этот вызов задает отображение числа байтов, определённых **len**, для файла с дескриптором **fildes**, начиная со смещения **off** в область памяти со стартовым адресом **addr**. Перед вызовом **mmap** файл должен быть открыт с помощью функции **open**.

Аргумент **prot** определяет права доступа к области памяти, которые должны соответствовать правам доступа к файлу. Значение **len** ОС округляет до границы следующей страницы виртуальной памяти (выделяется область, кратная странице).

Использование права на выполнения, позволяет процессу определить собственный механизм загрузки кода. Такой подход используется редактором динамических связей, при загрузке динамических библиотек, когда библиотека отображается в адресное пространство процесса.

Аргумент **flags** определяет дополнительные особенности управления памятью и может иметь следующие значения:

MAP\_SHARED ⎯ область памяти может использоваться несколькими процессами;

MAP\_PRIVETE ⎯ область памяти используется только вызывающим процессом;

MAP\_FIXED ⎯ требуется выделение памяти начиная точно с **addr;**

MAP\_NORESERVED ⎯ не требуется резервировать область свопинга.

*#include <sys/types.h>*

*#include <sys/stat.h>*

*#include <unistd.h>*

*#include <fcntl.h>*

*main (int argc, char \*argv[])*

*{*

*int fd\_src, fd\_dst;*

*caddr\_t addr\_src, addr\_dst;*

*struct stat filestat;*

*fd\_src=open(argv[1],O\_RDONLY);*

*fd\_dst=open(argv[2],O\_RDWR|O\_CREAT);*

*fstat(fd\_src, &filestat);*

*lseek(fd\_dst, filestat.st\_size-1,SEEK\_SET);*

*write(fd\_dst,“ ”,1);*

*addr\_src=mmap((caddt\_t)0,filesta.st\_size,PROT\_READ,MAP\_SHARED, fd\_src,0));*

*addr\_dst=mmap((caddr\_t)0, filestat.st\_size,PROT\_READ|PROT\_WRITE, MAP\_SHARED, fd\_dst);*

*memcpy(addr\_dst, addr\_src, filestat.st\_size);*

*exit(0);*

*}*

**Каталоги**

Права доступа к каталогам устанавливаются так же как и к обычным файлам, но их интерпретация несколько другая. Право доступа для чтения к каталогу показывает, что соответствующая категория пользователей может выводить список содержащихся в каталоге файлов, но это не значит, что эта категория может читать информацию в файлах, которые записаны в этом каталоге, потому что доступ к файлу регулируется правами доступа к файлу. Права доступа к каталогу на запись позволяет создавать новые файлы и удалить существующие. Но менять содержимое файлов можно только на основании прав доступа владельца. Права доступа к каталогу на выполнение позволяет перейти в каталог с помощью команды cd или системного вызова chdir. А для того, чтобы открыть файл или выполнить программу, пользователь должен иметь право доступа на выполнение для всех ведущих каталогов, входящих в эту группу. Если для каталога установлен вид фиксации, то пользователь может удалить из каталога только те файлы, которыми он владеет или по отношению к которым он имеет право на запись даже при наличии права на запись в каталоге.

*Прототипы*:

*#include <sys/types.h>*

*#include <dirent.h>*

*int mkdir (const char \*pathname, mode\_t mode); - созданиекаталога*

*int rmdir (const char \*pathname); - удалениекаталога*

*DIR \*opendir(const char \*dirname); - открытиекаталога*

*int closedir(DIR \*dirptr); - закрытиекаталога*

*struct dirent \*readdir (DIR \*dirptr); - чтениеочереднойзаписи*

*voidrewinddir(DIR \*dirptr); - возвращение в начало списка записей*

*int chdir(const char \*path); - сменакаталога*

*char \*getcwd(char \*name, seze\_t size); - определениетекущегокаталога.*

*Примеры:*

*/\* функция двойного вывода содержимого каталога\*/*

*int double\_ls\_dir(const char \*name)*

*{ struct dirent \*d;*

*DIR \*dp;*

*if ((dp=opendir(name))==NULL) return (-1);*

*while (d=readdir(dp))*

*{ if (d->inv!=0) printf(“%s\n”,d->d\_name); }*

*rewinddir(dp);*

*while (d=readdir(dp))*

*{ if (d->inv!=0) printf(“%s\n”,d->d\_name); }*

*close(dp);*

*return(0);*

*}*

*/\* функция поиска файлов, заканчивающих на заданное расширение \*/*

*#include <stdio.h>*

*#include <dirent.h>*

*#include <string.h>*

*int match(const char \*s1, const char \*s2)*

*{ int diff;*

*dif=strlen(s1)-strlen(s2);*

*if (strlen(s1)>strlen(s2)) return (strcmp(&s1[diff],s2==0)*

*else retunr(0);*

*}*

*char \*find\_entry(char \*dirname, char \*suffix, int cont)*

*{ static DIR \*dp=NULL;*

*struct dirent \*d;*

*if ((dp==NULL)||(cont=0))*

*if (dp=opendir(dirname)==NULL) return(NULL);*

*while (d=readdir(dp))*

*{*

*if (d->d\_inv==0) continue;*

*if (match(d->d\_name,suffix)) return (d->d\_name);*

*}*

*closedir(dp);*

*dp=NUL;*

*return(NULL);*

*}*

**Метаданные файлов**

Каждый файл имеет, помимо своих собственных данных, метаданные, которые описывают его характеристики. Метаданные файла хранятся в структуре индексного дескриптора. Часть полей этой структры можно получить с помощью следующих системных вызовов.

*#include <sys/types.h>*

*#include <sys/stat.h>*

*int stat (const char \*path, struct stat \*buf);*

*int lstat (const char \*path, struct stat \*buf);*

*int fstat (int fildes, struct stat \*buf);*

В качестве аргумента функции принимают имя файла или файловый дескриптор и возвращают заполненные поля структуры stat, которые приведены ниже.

* *mode\_t st\_mode* - тип файла и права доступа;
* *ino\_t st\_ino* - номер дескриптора или структуры (inode), где хранятся метаданные данного файла;
* *dev\_t st\_dev -* идентификатор устройства, содержащего файл;
* *dev\_t st\_rdev*- для специального файлового устройства, содержит номер устройства, адресуемого этим файлом;
* *nlink\_t st\_nlink* - число жестких связей файла;
* *uid\_t st\_uid* - идентификатор пользователя-владелеца файла;
* *gid\_t st\_gid* - идентификатор группы владельца файла;
* *off\_t st\_size* - размер обычного файла в байтах; для специального файла устройств это поле не определено;
* *time\_t st\_atime* - время последнего доступа к файлу;
* *time\_t st\_mtime* - время последней модификации файла;
* *time\_t st\_ctime* - время последнего изменения метаданных файла;
* *long st\_blksize* - оптимальный размер блока для операций ввода/вывода; для специальных файлов это поле не определено;
* *long st\_blocks* - число размещенных 512 байтовых блоков, которые выделены для хранения данных файла.

Для определения типа файла используются специальные макроопределения:

S\_ISFIFO(mode) - FIFO

S\_ISCHR(mode) - символьноеустройство

S\_ISDIR(mode) - каталог

S\_ISBLK(mode) - блочное устройство

S\_ISREG(mode) - обычный файл

S\_ISLNK(mode) - символическая связь

S\_ISSOCK(mode) - сокет

Существуют функции, позволяющие изменить владельца файла и владельца группы, а также изменить права доступа к файлу.

**Базовая файловая система UNIX s5fs**

2.1. Структура файловой системаUNIXs5fs

Каждый жесткий диск состоит из одного или нескольких разделов. Расположение и размер разделов определяется при форматировании диска. Разделы выступают в качестве независимых устройств, доступ к которым осуществляется как к различным носителям данных. В разделе может располагаться только одна файловая система, которая не может занимать нескольких разделов.

Файловая система s5fs занимает раздел диска и имеет следующую структуру:

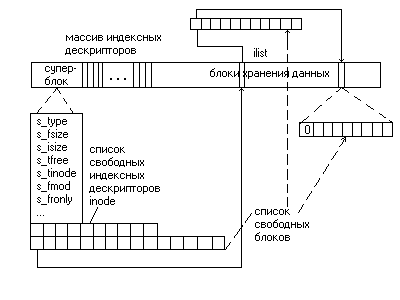


Рис. 2.6. Структура файловой системы s5fs

Суперблок содержит общую информацию о файловой системе, необходимую для монтирования и управления файловой системой в целом, например, для размещения новых файлов. В каждой файловой системе существует только один суперблок, который размещается в начале раздела. Суперблок считывается в память при монтировании файловой системы и находится там до ее размонтирования. Суперблок содержит следующую информацию:

- тип файловой системы;

- размер файловой системы в логических блоках, включая сам суперблок, массив файловых дескрипторов и блоки хранения данных;

- размер массива индексных дескрипторов;

- число свободных блоков, доступных для размещения;

- число свободных индексных дескрипторов, доступных для размещения;

- флаг модификации;

- флаг режима монтирования;

- размер логического блока (512, 1024, 2048);

- список номеров свободных индексных дескрипторов;

- список адресов свободных блоков.

Так как число свободных файловых дескрипторов и блоков хранения данных может быть большим, списки свободных индексных дескрипторов и и списки свободных блоков целиком в суперблоке не хранятся.

Для индексных дескрипторов хранится только часть списка. Когда число свободных индексных дескрипторов в этом списке приближается к нулю, ядро операционной системы просматривает массив индексных дескрипторов и вновь формирует список свободных индексных дескрипторов. Индексные дескрипторы имеют специальное поле, по которому можно определить, занят он или свободен.

По содержимому блоков хранения данных нельзя определить, занят этот блок или свободен, поэтому необходимо хранить список адресов свободных блоков целиком. Суперблок содержит только один блок из этого списка. Первый элемент этого блока указывает на блок, хранящий продолжение этого списка и т.д. Выделение свободных блоков для размещения файлов производится с конца списка суперблока. Когда в списке остается единственный элемент, ядро интерпретирует его как указатель на блок, содержащий продолжение этого списка. В этом случае содержимое этого блока считывается в суперблок и блок становится свободным. Такой подход позволяет использовать дисковое пространство под списки, пропорциональное свободному месту в файловой системе, то есть когда свободного места становится мало, список адресов свободных блоков помещается целиком в суперблоке.

2.2. Индексные дескрипторы

Массив индексных дескрипторов содержит метаданные всех файлов файловой системы. Индексный дескриптор содержит информацию о файле, необходимую для обработки данных, хранящихся в файле. Каждый файл ассоциирован с одним индексным дескриптором, хотя в файловой системе может быть несколько имен, каждый из которых указывает на один и тот же индексный дескриптор. Ядро обращается к индексному дескриптору по индексу в массиве индексных дескрипторов. Один индексный дескриптор является корневым индексным дескриптором файловой системы. Через него обеспечивается доступ к структуре каталогов и файлов после монтирования файлов системы. Размер массива индексных дескрипторов является фиксированным и задается при создании файловой системы. Следовательно, файловая система s5fs имеет ограничение по числу файлов, которые могут в ней храниться, независимо от размера этих файлов.

Индексный дескриптор имеет следующую структуру:

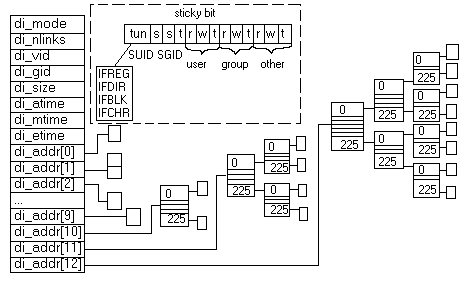


Рис. 2.7. Структура индексного дескриптора

Индексный дескриптор не содержит имени файла, которое хранится в блоке хранения данных каталога, и содержимого файла, которое размещено в блоках хранения данных. При открытии файла ядро помещает копию дискового индексного дескриптора в память в специальную таблицу, которая содержит несколько дополнительных полей.

Основные поля индексного дескриптора следующие.

1. Поле di-mode хранит атрибуты файла, такие, как тип файла и права доступа к файлу, а также три бита под модификацию прав доступа. Типы могут быть следующими:

- обычные файлы;

- каталоги;

- специальные файлы блочных устройств;

- специальные файлы символьных устройств.

2. Число ссылок на файл (количество имен, которые имеет файл в файловой системе).

3. Идентификаторы владельца-пользователя и владельца-группы.

4. Размер файла в байтах. Для специальных файлов это поле содержит старший и младший номера устройства.

5. Время последнего доступа к файлу.

6. Время последней модификации.

7. Время последней модификации индексного дескриптора.

Далее - массив адресов дисковых блоков хранения данных. Массив имеет фиксированный размер и состоит из двенадцати элементов. Первые десять элементов адресуют непосредственно блоки хранения данных файла, одиннадцатый элемент адресует блок, который содержит адреса блоков хранения данных. Двенадцатый элемент указывает на дисковый блок, хранящий адреса блоков, каждый из которых адресует блок хранения данных файла (то есть, двойная косвенная адресация).

Тринадцатый элемент используется для тройной косвенной адресации, при его применении для нахождения адреса блока хранения данных файла используются три дополнительных блока.

2.3. Блоки хранения данных

Блоки хранения данных занимают большую часть дискового раздела и их число определяет максимальный суммарный объем файлов данной файловой системы. Размер блока выбирается кратным 512 байтам. Обработка данных файла осуществляется через индексный дескриптор, который содержит ссылки на блоки данных. Файлы в операционной системе UNIX могут содержать пустые зоны, или так называемые дыры. Процесс может создать пустой файл с помощью системного вызова lseek, сместив файловый указатель относительно начала файла и сделать запись данных, при этом между началом файла и началом записанных данных образуется незаполненная область, при чтении этой области процесс получит нулевые байты. Так как логические блоки, соответствующие дыре, не содержат данные, нет смысла размещать для них дисковые блоки. В этом случае соответствующие элементы массива адресов индексных дескрипторов содержат нулевой указатель.

Когда процесс производит чтение такого блока, ядро возвращает последовательность нулей. Дисковые блоки размещаются для таких файлов только при записи в соответствующие логические блоки файла.

2.4. Имена файлов

Имя файла хранится в файлах специального типа, называемых каталогами, но ни метаданные, ни блоки хранения данных не содержат имени файла. Такой подход позволяет файлу иметь практически неограниченное число имен.

Каталог файловой системы s5fs представляет собой таблицу, каждый элемент которой имеет фиксированный размер в 16 байтов. Два байта хранятся в номере индексного дескриптора, а 14 байтов – его имя. Таким образом, число индексных дескрипторов не может превышать 65535. Первый элемент каталога адресует сам текущий каталог и имеет имя “.” (точка), а второй – родительский каталог под именем “..” (две точки). При удалении имени файла из каталога номер индексного дескриптора соответствующего элемента устанавливается равным нулю. Обычно ядро не удаляет такие свободные элементы, поэтому размер каталога не уменьшается даже при удалении файла. Это составляет определенные проблемы для каталогов с большим числом файлов.

2.5. Недостатки и ограничения файловой системы s5fs

1. С точки зрения надежности слабым местом в файловой системе является суперблок, так как при его повреждении файловая система не может использоваться. Так как суперблок хранится в единственном экземпляре, вероятность появления ошибок достаточно высокая.

2. Относительно низкая производительность связана с размещением файловых компонентов на диске: индексный дескриптор размещается в начале файловой системы, а блоки хранения данных – на остальном пространстве диска, а при работе с файлами происходит как обращение к метаданным, как и блокам данных файла, и так как эти элементы могут быть достаточно разнесены по файловому пространству, необходимо перемещать головки диска, что увеличивает время.

3. Использование дискового пространства не оптимально. Для повышения производительности файловой системы желательно использовать блоки большого размера, но если файлы имеют небольшой размер и их количество значительно, получаются значительные потери дискового пространства из-за недозаполнения блоков.

4. Массив индексных дескрипторов имеет фиксированный размер, задаваемый при создании файловой системы. Этот размер накладывает ограничения на максимальное число файлов, которые могут существовать в файловой системе.

Ограничение на длину имени файла - 14 символов. Максимальное число индексных дескрипторов 65535.

**ФайловаясистемаFFS (FastFileSystem)**

3.1. Структура файловой системы FFS

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Супер-блок | Массив свободных блоков и индексных дескрипторов | Массив индек-сных дес-крипто-ров | Блоки хране-ния дан-ных | Супер-блок | Массив свободных блоков и идексных дескрип-торов | Массив индек-сных дес-крипто-ров | Блоки хране-ния дан-ных | Супер-блок |

Рис. 2.8. Структура файловой системы FFS

Файловая система FFS обладает полной функциональностью системы s5fs и использует те же структуры данных ядра ОС. Отличия касаются расположения файловой системы на диске, дисковых структур данных и алгоритмов размещения свободных блоков.

Здесь суперблок также содержит полное описание файловой системы и располагается в начале раздела, но в нём не хранятся данные о свободном пространстве файловой системы, т.е. дескрипторы свободных блоков и индексные дескрипторы. Следовательно данные суперблока остаются неизменными на протяжении существования файловой системы. Для повышения надёжности файловой системы суперблок дублируется.

Организация файловой системы предусматривает логическое деление дискового раздела на одну или несколько групп цилиндров. Группа цилиндров -несколько последовательных дисковых цилиндров. Каждая группа цилиндров содержит управляющую информацию, включающую резервную копию суперблока, массив индексных дескрипторов, данные о свободных блоках, информацию об использовании дисковых блоков в группе. Для каждой группы цилиндров при создании файловой системы выделяется место под определенное количество индексных дескрипторов. Обычно на каждые два килобайта блоков хранящихся данных создается один индексный дескриптор. Следовательно, есть ограничение не общее количество файлов, которое может быть записано в одной группе цилиндров.

В этой файловой системе кластеры индексных дескрипторов располагаются по всему дисковому разделу, а не в его начале. За счёт этого достигается уменьшение время доступа к данным файла, т.к. блоки данных располагаются ближе к описывающим их индексным дескрипторам. Это также повышает надёжность файловой системы, т.к. уменьшается вероятность потери всех индексных дескрипторов в случае сбоя.

Управляющая информация располагается с различными смещениями от начала группы цилиндров. Это ведёт к тому, что управляющая информация не располагается на одной дисковой пластине, иначе выход из строя одной пластины привел бы к выходу из строя всей файловой системы.

Производительность файловой системы существенно зависит от размера блока хранения данных. Чем больше размер блока, тем больше данных может быть прочитано без поиска и перемещения дисковых головок. FFS поддерживает размер блока до 64Kb. Для преодоления проблемы потери дискового пространства вводится возможность фрагментации дискового блока. Каждый блок дополнительно разбивается на 2, 4 или 8 фрагментов. Блок является единицей передачи данных в операциях ввода/вывода, а фрагмент определяет адресуемую единицу хранения данных на диске. Минимальный размер сегмента определяется размером сектора.

Информация о свободных блоках и фрагментах хранится виде битовой карты блоков. Карта блоков, связанная с определенной группой цилиндров, описывает свободное пространство в фрагментах. Для определения свободен фрагмент или занят, ядро ОС анализирует биты карты.

Файловая система FFS при размещении блоков использует стратегию, направленную на увеличение производительности. Основные принципы этой стратегии таковы.

1. Файл по возможности размещается в блоках хранения данных, принадлежащих одной группе цилиндров, где расположены его метаданные.

2. Все файлы каталога, по возможности, размещаются в одной группе цилиндров.

3. Каждый новый каталог, по возможности, помещается в группу цилиндров, отличную от группы родительского каталога. Этим достигается равномерное распределение данных по диску.

4. Последовательные блоки размещаются, исходя из оптимизации физического доступа.

Существует определенный момент времени между завершением чтения блока и моментом чтения следующего блока. За это время диск совершает поворот на некоторый угол. Таким образом, следующий блок для чтения должен, по возможности, располагаться с пропуском нескольких секторов. В этом случаи не требуется совершать пустые обороты диска, т.е. правила размещения свободных блоков на диске направлены, с одной стороны, на уменьшение времени перемещения головок и пустых оборотов дисков, а с другой стороны - на равномерное распределение данных по диску.

Производительность данной файловой системы хороша при не слишком большой загрузке диска и ухудшается при малом свободном пространстве (менее 10% объёма).

3.2. Каталоги FFS

Структура каталогов файловой системы FFS изменена для обеспечения поддержки длинных имен файлов (до 255 символов). Вместо таблиц, представляющих каталоги, здесь используется список структур, которые имеют следующий формат:

|  |  |  |  |
| --- | --- | --- | --- |
| 137 | | | |
| 4 | | | |
| 1 | | | |
| ‘.’ | 0 | 0 | 0 |
| 237 | | | |
| 4 | | | |
| 2 | | | |
| ‘.’ | ‘.’ | 0 | 0 |
| 945 | | | |
| 12 | | | |
| 9 | | | |
| ‘f’ | ‘i | ‘l’ | ‘e’ |
| ‘-‘ | ‘p’ | ‘r’ | ‘o’ |
| ‘g’ | 0 | 0 | 0 |

d\_ino - номер индексного дескриптора;

d\_reclen - длина записи;

d\_namelen - длина имени файла;

d\_name[] - имя файла.

Имя файла имеет переменную длину, дополняемую нулями до 4-х байтной границы. При удалении имени файла значение первого поля устанавливается в 0, а освободившееся число байт прибавляется к полю **d\_reclen** предыдущей записи.

***Тема 3. Процессы в ОС UNIX***

**Место процессов в ОС UNIX**

В ОС UNIX процессы занимают одно из важнейших мест. От оптимальной настройки системы управления процессами и числа одновременно выполняемых процессов зависит загрузка процессора, что влияет на общую производительность системы.

Ядро ОС предоставляет задачам базовый набор услуг, определяемых базовым интерфейсом системных вызовов. К ним относятся основные операции работы с файлами, управления процессами и памятью, поддержка межпроцессорного взаимодействия.

Под программой можно понимать совокупность файлов (тексты программы, объектные файлы, выполняемые файлы). Перед запуском программы на выполнение ОС должна создать окружение (среду выполнения задачи). Сюда относятся ресурсы памяти и возможность доступа к устройствам ввода/вывода и различным системным ресурсам, включая услуги ядра.

Процесс можно представить как совокупность данных ядра системы, необходимых для описания программы в памяти и управления её выполнением, или как программу, на стадии выполнения, т.к. все выполняющиеся программы предоставлены в ОС UNIX в виде процессов.

Процесс состоит из инструкций, выполняемых процессором, данных и информации о выполняемой задаче. Процесс не отождествляется с программой, т.к. программа может породить более одного процесса.

Выполнение процесса заключается в точном следовании набору инструкций, который никогда не передаёт управление другому процессу. Процессу также недоступны данные и стеки других процессов. Но процессы могут обмениваться друг с другом данными с помощью системы межпроцессорного взаимодействия, предоставляемой ОС UNIX. К этой системе относятся сигналы, каналы, разделяемая память, семафоры, сообщения, файлы. В остальном процессы изолированы.

**Типы процессов**

.1. Системные процессы

Системные процессы являются частью ядра ОС и всегда находятся в оперативной памяти. Они не имеют соответствующих программ в виде исполняемых файлов и запускаются особым образом при инициализации ядра системы. Инструкции и данные этих процессов находятся в ядре системы, и они могут обращаться к функциям и данным, недоступным для остальных процессов.

Системными процессами являются:

* shed - диспетчер свопинга;
* vhand - диспетчер страничного замещения;
* bdfflush - диспетчер буферного кэша;
* kmadaemon - диспетчер памяти ядра.

К системным процессам также можно отнести процесс init (хотя он не является частью ядра и его запуск производится из специального файла в /etc) являющийся прародителем всех остальных процессов.

.2. Демоны

Демоны - не интерактивные процессы, которые запускаются обычным образом (загрузка соответствующей программы) и выполняются в фоновом режиме. Обычно демоны запускаются при инициализации системы и обеспечивают работу различных подсистем (терминального доступа, печати, сетевого доступа и т.д.)

Демоны не связаны ни с одним пользовательским сеансом и не могут непосредственно управляться пользователем. Большую часть времени демоны находятся в состоянии ожидания, пока какой-нибудь процесс не запросит определённую услугу.

.3. Прикладные процессы

К ним относятся все остальные процессы, выполняющиеся в системе. В большинстве случаев - это процессы, порожденные в рамках пользовательского сеанса работы.

Важнейший пользовательский процесс - командный интерпретатор (shell), который запускается после регистрации пользователя в системе, а его завершение приводит к отключению пользователя от системы.

Пользовательские процессы могут выполняться как в интерактивном, так и в фоновом режиме, но время их жизни ограничено временем работы пользователя. При выходе из системы все пользовательские процессы уничтожаются.

Интерактивные процессы монопольно владеют терминалом. И пока такой процесс не завершит работу, пользователь не может работать с другими процессами, если в функции интерактивного процесса не входит запуск на выполнение других программ.

**Создание процесса**

Когда интерпретатор shell запускает новую программу, осуществляется порождение нового процесса, а затем загружается требуемая программа, и операционная система, соответственно, предоставляет для этих действий два системных вызова. Первый - для создания процесса, а другой - для запуска новой программы.

Для порождения процесса имеется специальный системный вызов **fork**.

*Прототип:*

*#include <sys/types.h>*

*#include <unistd.h>*

*pid\_t fork(void);*

*Пример:*

*/\* создаёт один дочерный процесс \*/*

*main( )*

*{pid\_tpid;*

*printf(“Функционирует один процес\n”);;*

*pid = fork( );*

*if (pid == 0) printf(“\nВыполняется процесс-сын\n”);*

*elseif (pid> 0) printf(“\nВыполняется процесс-отец\n”);*

*else printf(“Ошибка при создании процесса\n”);*

*}*

Процесс до выполнения fork()

Процесс-сын

Процесс-отец

Fork>0

Fork == 0

Новый процесс порождается с помощью системного вызова **fork**. Порожденный процесс (процесс-сын) является точной копией процесса, выполнившего этот системный вызов, то есть родительского процесса. Процессу сыну передаются следующие атрибуты родительского процесса:

- идентификаторы пользователя и группы;

- переменные окружения;

- диспозиция сигналов и их обработчики;

- текущий и корневой каталог;

- маска создания файлов;

- все файловые дескрипторы, включая файловые указатели;

- управляющий терминал.

Виртуальная память процесса-сына не отличается от образа родительской. Такие же сегменты кода, данных, стека, разделяемой памяти и других видов памяти.

После вызова **fork** уже развиваются два процесса. Родительский и дочерний процесс, оба выполняют одну и ту же команду, но по разным ветвям. Между порождающим и порожденным процессом имеются отличия:

- процессу-сыну присваивается уникальный идентификатор PID;

- родительские идентификаторы для процесса-отца и процесса-сына различны - у процесса-сына есть собственный идентификатор процесса-отца;

- значение, возвращаемое системным вызовом fork, различно для процесса-отца и процесса-сына. Отцу возвращается значение, равное идентификатору PID сына, а процессу-сыну возвращается значение 0. При ошибке возвращается значение –1.

При этом могут обнаруживаться две ошибочные ситуации, которые обнаруживаются в специальной переменной errno.

ENOMEM ⎯ для создания нового процесса не хватает памяти.

EAGAIN ⎯ количество текущих процессов превышает установленные в системе ограничения.

**Атрибуты процесса**

1. Идентификатор процесса (**PID**). Каждый процесс имеет уникальный идентификатор, позволяющий ядру системы различать процессы. Когда создается новый процесс, ядро присваивает ему следующий свободный идентификатор. Присваивание идентификаторов происходит по нарастанию, т.е. идентификатор следующего процесса больше идентификатора процесса, созданного перед ним. Если идентификатор достигает максимально возможного значения, то следующий процесс получит минимальный свободный идентификатор и цикл повторится. Когда процесс завершает свою работы, ядро освобождает занятый им идентификатор.

* *pid\_t getpid (void)* - системный вызов, возвращающий идентификатор данного процесса.

*Пример:*

*/\**

*Создаёт временный файл : /tmp/tmp<pid>.<n>, где*

*<pid> - идентификатор процесса*

*<n> - номер вызова функции*

*\*/*

*#include <string.h>*

*#include <unistd.h>*

*int num=0;*

*char namebuf[20];*

*char prefix[]=”/tmp/tmp”;*

*char \*gentemp(void)*

*{*

*int length;*

*pid\_t pid;*

*pid=getpid();*

*strcpy(namebuf,prefix);*

*length=strlen(namebuf);*

*itoa(pid,&namebuf[length]);*

*strcat(namebuf,”.”);*

*length=strlen(namebuf);*

*do{*

*itoa(num++,&namebuf[length])l;*

*}while (access(namebuf,F\_OK)!=-1)l*

*return (namebuf);*

*}*

*int itoa(int i; char \*string)*

*{*

*int power,j;*

*j=i;*

*for (power=1;j>=10;j/=10) power\*=10;*

*for (;power>j;power/=10)*

*{*

*\*string++=’0’+i/power;*

*i%=power;*

*}*

*\*string=’\0’;*

*}*

2. Идентификатор родительского процесса (**PPID**) - идентификатор процесса, породившего данный процесс.

* *pid\_t getppid (void)* - системный вызов, возвращающий идентификатор процесса-отца для текущего процесса.

3. Идентификатор группы процесса (**PGID)**. Каждая группа имеет идентификатор группы. Процесс, идентификатор которого совпадает с идентификатором группы, называется лидером группы. Первоначально процесс наследует идентификатор группы во время выполнения системного вызова fork() или exec().

* *pid\_t getpgrp (void)* - системный вызов, возвращающий идентификатор группы процесса.
* *intsetpgid (pid\_tpid,pid\_tpgid)* – системный вызов для создания новой группы или присоединения к существующей.

*pid – идентификатор процесса*

*pgid – устанавливаемый идентификатор группы*

Если pid=0, то используется идентификатор вызывающего процесса. Если pid и pgid равны, то процесс становится лидером группы. Если pgid=0, то в качестве идентификатора группы используется значение pid. в случае ошибки функция возвращает -1.

4. Приоритет процесса (**NiceNumber**). Относительный приоритет процесса учитывается планировщиком задач при определении очерёдности запуска. Фактическое распределение процессорных ресурсов определяется приоритетом выполнения, зависящим от нескольких факторов (в том числе и от задания относительного приоритета). Относительный приоритет не изменяется системой на всём протяжении жизни процесса, но может быть изменён пользователем или администратором, а приоритет выполнения динамически обновляется ядром.

5. Терминальная линия (**TTY**) - терминал (псевдотерминал), ассоцииро-ванный с процессом. Демоны не имеют ассоциированного терминала.

6. Сеанс и идентификатор сеанса. Когда пользователи входят в систему, все процессы и группы процессов будут принадлежать к сеансу, связанному с их текущим терминалом. Сеанс представляет собой набор из одной группы переднего плана, использующий терминал, и 1 или более групп фоновых процессов.

*#include <string.h>*

*#include <unistd.h>*

* *pid\_t getsid (pid\_tpid)* - системный вызов, возвращающий идентификатор сеанса процесса.
* *pid\_tsetsid (void)* – системный вызов для создания нового сеанса.

Демон может задать для себя сеанс без упрявляющего терминала, переместившись в другой сеанс при помощи системного вызова setsid(). Если вызывающий процесс не являлся лидером группы процессов, то создаётся новая группа процессов и новый сеанс; идентификатор вызывающего процесса станет идентификатор созданного сеанса.

7. Переменные програмного окружения. Програмное окружение – набор строк, заканчивающихся нулевым символом, представляется в программе в виде массива указателей. Эти строки – переменные окружения. По умолчанию окружение процесса передаётся дочерным процессам

*Пример:*

*/\* вывод переменных окружения \*/*

*main( intargc, char \*\*argv,char \*\*envp)*

*{*

*while (\*envp) printf(“%s\n”,\*envp++);*

*}*

*#include <stdlib.h>*

*exter char \*\*environ*

* *char \*getenv(constchar \*name)* - получить значение переменной окружения.
* *intputenv(char \*string)* – изменить значение переменной окружения.

8. Текущий рабочий каталог, текущий корневой каталог. Текущий рабочий каталог наследуется во время создания процесса. Если процесс изменит свой рабочий каталог с помощью функции chdir, то текущий каталог родительского процесса не изменится. С каждым процессом дополнительно связан корневой каталог, который используется при поиске абсолютного пути. Первоначально корневой каталог наследуется от родительского процесса, но может быть изменен:

*#include <stdlib.h>*

* *int chroot(const char \*path)* – изменитькорневой каталог

9. Реальный (**RUID**) и эффективный (**EUID**) идентификаторы пользователя. Реальный идентификатор - идентификатор пользователя, запустившего процесс, а эффективный идентификатор используется для определения прав доступа процесса к системным ресурсам. Обычно, реальный и эффективный идентификаторы идентичны. Однако есть возможность задать процессу более широкие права, путём установки флага SUID. В этом случае процесс получает значение идентификатора владельца выполняемого файла.

10. Реальный (**RGID**) и эффективный (**EGID**) идентификаторы группы. Реальный идентификатор группы равен идентификатору первичной группы пользователя, запустившего процесс. Эффективный идентификатор служит для определения права доступа к системным ресурсам по классу доступа групп. Также можно его модифицировать с помощью SUID.

**Системные функции типа exec**

Существует группа системных функций, которые предназначаются для перезапуска программы.

*Прототипы:*

*#include<unistd.h>*

*int execl (char \*path, char arg0, …, char argn, (char\*)NULL);*

*int execv (char \*path, char argv[ ]);*

*int execle (char \*path, char arg0, …, char argn, (char\*)NULL, char \*envp[ ]);*

*int execve (char \*path, char argv[ ], char \*envp[ ]);*

*int execlp (char \*file, char arg0, …, char argn, (char\*)NULL);*

*int execvp (char \*file, char argv[ ]);*

*Дерево:*

execl

execle

execlp

execv

execvp

execve

Они различаются по форме передачи параметров, но в общем все похожи. Первый аргумент функции является либо полным путевым именем, либо именем файла программы, которую надо выполнить. Если вызов проходит успешно, данные и команды возвращающего процесса, хранящиеся в памяти, перекрываются данными и командами новой программы и процесс начинает выполнение программы с ее первых строк. После завершения выполнения новой программы код завершения процесса передается обратно родительскому процессу.

В отличие от **fork**, создающего порожденный процесс, который выполняется независимо от родительского, **exec** изменяет содержание выполняющегося процесса для выполнения другой программы. Вызов функции **exec** может быть неудачным. В случае ошибки возвращается значение –1 в вызывающий образ, а также обнаруживается ряд ошибок.

Буква ‘p’ в вызовах exec указывает, что если значение аргумента **file** не начинается с ’/’, то есть корневого каталога, то функции при поиске надлежащего выполнению файла будут использовать переменную среды PATH. Для всех остальных функций exec фактическим значением их первого аргумента должно быть путевое имя файла.

*Примеры:*

*/\* Создание дочерного процесса, который выводит*

*содержимое текущего каталога путём вызава программы ls \*/*

*#include <unistd.h>*

*main()*

*{*

*pid\_t pid;*

*switch (pid=fork())*

*{*

*case -1: printf(“Ошибкавызова fork\n”); exit(1); break;*

*case 0: execl(“bin/ls”,”ls”,”-l”,(char \*)0);*

*printf(“Ошибкавызова ecexl\n”); exit(2); break;*

*default: wait();*

*printf(“Программа ls завершена\n”); exit(0);*

*}*

*}*

*/\* программа - интерпретатор \*/*

*#include<iostream.h>*

*#include<stdio.h>*

*#include<stdlib.h>*

*#include<errno.h>*

*#include<sys/types.h>*

*#include<sys/wait.h>*

*#include<unistd.h>*

*int system (char \*cmd)*

*{*

*pid\_t pid;*

*int status;*

*switch (pid = fork( ))*

*{*

*case –1: return(-1);*

*case 0: execl (“/bin/sh”, ”sh”, ”-c”, cmd, 0);*

*printf (“|n Ошибка execl”);*

*exit (errno);*

*default: if (waitpid(pid, &status, 0)== (pid && WIFEXITED(status)) )*

*return (WEXITSTATUS(status));*

*return(-1);*

*}*

*}*

*int main ( )*

*{*

*int rc = 0;*

*char buf[256];*

*do*

*{*

*printf (“sh>”); fflush (stdout);*

*if (!gets (buf)) break;*

*rc = system (buf);*

*} while (!rc);*

*return (rc);*

*}*

**Системные функции типа wait**

*Прототипы:*

*#include <sys/waith>*

*pid\_t wait(int \*status\_p);*

*pid\_t waitpid(pid\_t child\_pid, unt \*status\_p, int options);*

Родительский процесс использует системные вызовы **wait** и **waitpid** для перехода в режим ожидания завершения порождённого процесса и для выборки его статуса завершения. Эти вызовы также освобождают ячейку таблицы процессов, соответствующую порождённому процессу.

Функция **wait** приостанавливает выполнение родительского процесса до тех пор, пока ему не будет послан сигнал, либо пока один из его порождённых процессов не завершится или не будет остановлен.

Если порождённый процесс уже завершился до вызова wait, то wait немедленно завершится и возвратит статус завершившегося процесса, который будет хранится в **status\_p**, а возвращённое значение будет иметь значение PID завершившегося процесса.

Если родительский процесс не имеет порождённых процессов, завершения которых он ожидает, или был прерван сигналом при выполнении wait, то wait возвратит значение равное -1, а в **errno** будет находится код ошибки. Если было порождено более одного процесса, то wait будет ожидать завершения любого из них.

*Пример:*

*/\* ожидание завершения дочерного процесса вызовом wait \*/*

*#include <sys/wait.h>*

*#include <unistd.h>*

*#include <stdlib.h>*

*main()*

*{ pid\_t pid*

*int status,exit\_status;*

*if ((pid=fork())<0)*

*{ printf(“Ошибка fork\n”); exit(1); };*

*if (pid==0)*

*{ sleep(10); exit(2); };*

*if ((pid=wait(&status))==-1)*

*{ printf(“Ошибка wait\n”); exit(3); }*

*if (WIFEXITED(&status)) // завершен ли системным вызовом \_exit*

*{*

*exit\_status=WEXITSTATUS(status);*

*printf(“Статусзавершенияпроцесса %d равен %d\n”,pid,exit\_status);*

*}*

*exit(0);*

*}*

Функция **waitpid** более сложная и универсальная. В этой функции можно указать, завершения какого из порождённых процессов следует ожидать.

Параметр **child\_pid** может содержать следующие значения:

* > 0 - ожидать завершения процесса с этим PID;
* -1 - ожидать завершения любого порождённого процесса;
* 0 - ожидать завершения любого порождённого процесса, принадлежащего к той же группе, что и родительский процесс;
* < -1 - ожидать завершения любого порождённого процесса, идентификатор группы которого равен абсолютному значению аргумента.

Третий аргумент (**option**) определяет модификацию вызова waitpid.

* *WNOHANG* - не блокирующий вызов, т.е. функция немедленно возвращает управление, если нет завершённого процесса, отвечающего критериям ожидания. В противном случае функция будет блокирующей.
* *WUNTRACED* - функция будет ожидать завершения порождённого процесса, остановленного механизмом управления заданиями, но о статусе которого не было ранее сообщено. Если status\_p равен NULL, то нет необходимости запрашивать статус завершения порожденного процесса. Если в **status\_p** хранится адрес переменной типа int, то в неё будет помещён статус завершения порождённого процесса.

Родительский процесс может проверить это значение с помощью нескольких макроопределений:

* *WIFEXITED* (\*status\_p) ⎯ возвращает ненулевое значение, если порождённый процесс был завершён с помощью \_exit, иначе возвращает ноль.
* *WEXITSTATUS* (\*status\_p) ⎯ возвращает код завершения порождённого процесса, присвоенного \_exit. Это макроопределение следует использовать только тогда, если WIFEXITED возвращает ненулевое значение.
* *WIFSIGNALED* (\*status\_p) ⎯ возвращает ненулевое значение, если порождённый процесс был завершён по получению прерывающего сигнала.
* *WTERMSIG* (\*status\_p) ⎯ возвращает номер сигнала, завершившего порождённый процесс, если WIFSIGNALED вернуло ненулевое значение.
* *WIFSTOPPED* (\*status\_p) ⎯ возвращает ненулевое значение, если порождённый процесс был остановлен механизмом управления заданиями
* *WSTOPSIG* (\*status\_p) ⎯ возвращает номер сигнала, завершившего порождённый процесс, если WIFSTOPPED вернуло ненулевое значение.

Если возвращаемое **wait** или **waitpid** значение положительное, то это - идентификатор порождённого процесса. Иначе возвращается - 1.Тогда errno может быть:

* *ECHILD* - для **wait** процесс не имеет порождённых процессов, завершение которого он ожидает, а для **waitpid** это означает, что либо значение **child\_pid** недопустимо, либо процесс не может находиться в состоянии, определённом в option.
* *EFAULT* - аргумент **status\_p** указывает на недопустимый адрес.
* *EINVAL* - значение option недопустимо.

*/\* ожидание завершения дочерного процесса вызовом waitpid \*/*

*#inckude <iostream.h>*

*#inckude <stdio.h>*

*#inckude <sys/types.h>*

*#inckude <sys/wait.h>*

*#inckude <unistd.h>*

*int main()*

*{ pid\_t child\_pid, pid; int status;*

*switch (child\_pid=fork())*

*{*

*case (pid\_t) -1 : printf(“|nОшибка fork”); break;*

*case (pid\_t) 0 : printf(“\nПроцесс-сынсоздан”); \_exit(15);*

*default : printf(\n“Процесс-отец %d”);*

*pid=waitpid(child\_pid, &status, WUNTRACED));*

*}*

*if ( WIFEXITED(status) )*

*printf(“\n Статусзавершения %d”, WEXITSTATUS(status));*

*else*

*if ( WIFSTOPPED(status) )*

*printf(“\n Номерсигнала %d”,WSTOPSIG(status));*

*else if ( WIFSIGNALED(status) )*

*printf(“\n Номерперехваченногосигнала %d”,WTERMSIG(status));*

*else printf(“\n Ошибка waitpid!”);*

*\_exit(0);*

*return(0);*

*}*

**Системныйвызов \_exit**

*Прототипы:*

*#include <unistd.h>*

*#include <stdlib.h>*

*void \_exit(int exit\_code);*

*void exit(int status);*

*int atexit(void(\*func)(void));*

Выполнение системного вызова \_exit приводит к освобождению сегмента данных, сегмента стека и закрытию всех открытых дескрипторов файлов для процесса, который вызвал **\_exit**;

Но запись в таблице процессов, в которой был зарегистрирован этот процесс, не удаляется, т.е. она ещё не может быть занята другим процессом. Процесс переходит в состояние “зомби”, т.к. его дальнейшее выполнение не планируется. Удалить запись может только родительский процесс с помощью вызова wait и waitpid.

Если процесс порождает сына и заканчивается раньше, чем процесс-сын, то системный процесс **init** становится управляющим для процесса-сына, и после его завершения удаляет запись о нём в таблице процессов.

Аргумент **\_exit** - код завершения процесса, причём родителю передаются только 8 младших бит. Нулевое значение свидетельствует об успешном завершении, ненулевое - о завершении процесса с ошибкой или по причине какой-либо ситуации.

Библиотечная функция exit является надстройкой над системным вызовом \_exit. Она дополнительно очищает буфер и закрывает все открытые потоки для вызывающего процесса. Затем она вызывает все функции, которые были зарегистрируемы с помощью функции atexit. После этого вызывается системный вызов \_exit.

**Обработка ошибок. Коды ошибок**

Все системные вызовы в случае ошибок возвращают -1. Введена внешняя переменная externerrno, которая в этом случае содержит код ошибки. Она не обнуляется успешно завершенными вызовами.

|  |  |
| --- | --- |
| Код | Значение |
| *EAGAIN* | *Ресурс временно недоступен (переполнена системная таблица)* |
| *EBADF* | *Недопустимый дескриптор файла* |
| *ECHILD* | *Нет доченрых процессов* |
| *EEXIST* | *Файл уже существует* |
| *EINVAL* | *Недопустимый аргумент* |
| *EIO* | *Ошибка ввода-вывода* |
| *EMFILE* | *Процессом открыто слишком много файлов* |
| *ENFILE* | *Переполнение в таблице открытых файлов* |
| *EINTR* | *Системный вызов прерван сигналом* |

*Прототипы:*

*include <string.h>*

*#include <errno.h>*

*#include <stdio.h>*

*char \*strerror(int errnum);*

*void perror(const char \*s);*

Функция strerror принимает код ошибки и возвращает указатель на строку, в которой находится сообщение об ошибке.

*Пример:*

*#include <errno.h>*

*#include <stdio.h>*

*main(int argc,char \*argv[])*

*{*

*fpritf(stderr,”ENOMEM:%s\n”,strerror(ENOMEM);*

*errno=ENOEXEC;*

*perror(argv[0]);*

*}*

***Тема 4.Взаимодействие между процессами***

**Взаимодействие процессов через файлы**

Все файлы, открытые в родительском процессе до системного вызова fork(), будут открыты в процессе-сыне. Процесс-сын будет иметь свою копию свызанных с файлами дескрипторами. Но указатель для чтения-записи для каждого из таких файлов используется совместно процессами. Если процесс-сын выполняет перезагрузку программы с помощью exec(), то указатели чтения-записи остаются неизменными и после выполнения exec(). Существует специальный флаг, при установке которого при вызове exec() все файловые дескрипторы закрываются.

*Пример:*

*#include <unistd.h>*

*#include <fcntl.h>*

*int fatal(char \*s)*

*{*

*printf(“%s\n”,s);*

*exit(1);*

*}*

*int print\_pos(const char\*string,int fd)*

*{*

*off\_t pos;*

*if ((pos=fseek(fd,0,SEEC\_CUR))==-1) fatal(“Ошибкавызова lseek”);*

*printf(“%s%ld\n”,string,pos);*

*}*

*void main()*

*{*

*int fp;*

*pid\_t pid;*

*char buf[10];*

*if ((fd=open(“data”,O\_RDONLY))==-1 fatal(“Ошибкавызова open”);*

*read(fd,buf[10])*

*print\_pos(“Довызова fork”,fd);*

*switch (pid=fork())*

*{*

*case -1 : fatal(“Ошибкавызова fork”); break;*

*case 0 : print\_pos(“Процесс-сын до чтения”,fd);*

*read(fd,buf,10);*

*print\_pos(“Процесс-сын после чтения”); break;*

*default: wait();*

*print\_pos(“Процесс-отец после завершения процесса-сына”,fd);*

*}*

*}*

**Взаимодействие процессов через каналы**

*Прототип:*

*#include <unistd.h>*

*int pipe(int fifo[2]);*

Для создания коммуникационного канала между процессами используется системный вызов **pipe**, который создаёт файл канала, служащий временным буфером и использующийся для того, чтобы процесс мог записывать туда данные для других процессов и читать данные для себя. Файлу канала имя не присваивается, следовательно это “неименованный канал”. Канал освобождается после того, как все процессы закроют дескриптор, ссылающийся на этот канал.

Аргумент pipe - массив из двух целых чисел.

В большинстве UNIX-систем каналы однонаправленные, т.е. для чтения данных из канала используется **fifo[0]**, а для записи - **fifo[1]**

К данным, хранящимся в канале, доступ производится последовательно по принципу First In First Out (FIFO). Процесс не может использовать функции позиционирования в канале. Данные удаляются из канала сразу после считывания.

Fifo[1]

Fifo[0]

Fifo[1]

Fifo[0]

Процесс 1

Процесс 2

Можно определить две типовые схемы взаимодействия.

1. Родитель - сын: родитель вызывает pipe для создания канала, а затем с помощью fork порождает сына. Т.к. порожденный процесс имеет копию дескрипторов файлов процесса-отца, то теперь процесс-отец и процесс-сын могут взаимодействовать через **fifo[0]** и **fifo[1]**

2. Брат - брат: родитель вызывает pipe, а затем порождает 2 или более сыновей-братьев. Процессы-браться взаимодействуют между собой через **fifo[0]** и **fifo[1]**

*Пример:*

*/\* родительский процесс читает сообщения порожденного*

*процесса-сына с помощью дескриптора fifo \*/*

*#include <stdio.h>*

*#include <unistd.h>*

*#defin MSG\_SIZE 9*

*char \*msg1=”Данные 1”;*

*char \*msg2=”Данные 2”;*

*int main()*

*{ pid\_t pid;*

*int fifo[2],j;*

*char inbuf[MSG\_SIZE];*

*if ( pipe(fifo)==-1) printf (“\nОшибка pipe!”); \_exit(1);*

*switch(pid=fork())*

*{*

*case -1 : printf (“\n Ошибка fork!”); \_exit(2);*

*case 0 : close(fifo[0]);*

*write (fifo[1],msg1,MSG\_SIZE);*

*write (fifo[1],msg1,MSG\_SIZE);*

*close(fifo[1]);*

*default: close(fifo[1]);*

*for (j=0;j<2;j++)*

*{ read(fifo[0],inbuf,MSG\_SIZE); printf(“ %s \n”,inbuf); }*

*close(fifo[0]);*

*}*

*exit(0);*

*}*

Буфер, связанный с каналом, имеет конечный размер. Минимальный размер канала – 512 байт. При попытке записать данные в уже заполненный канал, ядро заблокирует процесс до тех пор, пока другой процесс не считает из канала достаточное количество данных, чтобы заблокированный процесс смог возобновить запись. Таким образом ядро системы пытается организовать в канал запись данных неделимой порцией.

*Пример*

*/\* проверка ограниченности размера канала \*/*

*#include <unistd.h>*

*#include <signed.h>*

*#include <limits.h>*

*int count;*

*void alrm\_action(int signo)*

*{*

*printf(“Запись заблокирована после вывода %d символов\n’,count);*

*exit(0);*

*}*

*main()*

*{ int p[2],pipe\_size;*

*char c=’x’;*

*static struct sigaction act;*

*act.sg\_handler=alarm\_action;*

*sigfillset(&(act.sg\_mask));*

*sigaction(SIGALRM,&act,NULL);*

*if (pipe(p)==-1) { printf(“Ошибка pipe!\n”); exit(1); }*

*pipe\_size=fpathconf(p[0],\_PC\_PIPE\_BUF);*

*printf(“Максимальный размер канала %d байт\n”,pipe\_size);*

*while (1)*

*{*

*alarm(20);*

*write(p[1],&c,1);*

*alarm(0);*

*if ((++count%1024)==0) printf(“Вканале %d символов\n”,count);*

*}*

*}*

В этом примере использовалась функция *fpathconf*, с помощью которой можно получить ограничения для переменной или объекта, при этом второй аргумент указывает на тип ограничения.

Количество процессов, которые могут присоединиться к каналу, принципиально не ограничено. Однако если два или более процессов записывают в канал данные, то величина записи каждого из процессов ограничена определенной в системе величиной. Через некоторое время найдется процесс, считывающий данные из канала, после этого опять очередь перейдёт к данному процессу и он дозапишет информацию.

Для предотвращения такого недостатка коммуникационный канал обычно делают однонаправленным и используют его для связи между двумя процессами, причем один процесс является отправителем, а другой ⎯ получателем. Если оба процесса требуют при взаимодействии как операции передачи, так и операции приема, то целесообразно для связи между ними создать два канала.

F1[1]

F2[0]

F1[0]

F2[1]

Процесс 1

Процесс 2

Если дескриптора файла, связанного с записывающей стороной канала не существует, то эта сторона считается закрытой и любой процесс, пытающиеся считать данные из канала, получает оставшиеся в нем данные. Если все данные прочитаны, то получает признак конца файла. Если же не существует дескриптора файла, связанного с читающей стороной канала, а процесс пытается записать данные в канал, то ядро формирует для данного канала сигнал SIGPIPE. Действие по умолчанию на этот сигнал - завершение процесса.

Если дескриптор для записи в канал установлен неблокирующим, то системный вызов записи write не будет блокироваться, а будет возвращать -1 и errno::=EAGAIN. Если дескриптор для чтения с канала установлен неблокирующим, то системный вызов чтения read в случае пустого канала возврати -1 и errno::=EAGAIN.

*Пример:*

*/\* организация неблокирующего действия чтения-записи \*/*

*#include <unistd.h>*

*#include <fcntl.h>*

*#include <errno.h>*

*#defin M\_S 10*

*char \*msg1=”Начало!!!”;*

*char \*msg2=”Окончание”;*

*int parent(int p[2])*

*{ int n\_read;*

*char buf[M\_S];*

*close[p[1]);*

*for(;;)*

*{*

*switch (n\_read=read(p[0],buf,M\_S))*

*{*

*case -1 : if (errno==EAGAIN)*

*{ printf(“Каналпуст\n”); sleep(1); break; }*

*else { printf(“Ошибка read\n”); exit(1); }*

*case 0 : printf(“Канал закрыт. Конец связи\n”); exit(0);*

*default: printf(“MSG=%s\n,buf);*

*}*

*}*

*int child(int p[2])*

*{ int count;*

*close(p[0]);*

*for (count=0;count<3; count++)*

*{*

*write(p[1],msg1,M\_S);*

*sleep(3);*

*}*

*write(p[1],msg2,M\_S);*

*exit(0);*

*}*

*int main()*

*{ int pfd[2],j;*

*if ( pipe(fifo)==-1) printf (“\nОшибка pipe!”); \_exit(1);*

*if (fcntl(pfd[0],F\_SETFL,O\_NONBLOCK)==-1)*

*{ printf(“Ошибка fcntl\n”); exit(2); }*

*switch (fork())*

*{*

*case -1 : printf (“\n Ошибка fork!”); \_exit(3);*

*case 0 : child(pfd);*

*defaut : paret(pfd);*

*}*

*}*

*Прототип:*

*#include <sys/time.h>*

*int select (int nfds, fd\_set \* readfds, fd\_set \* writefds, fd\_set \* errorfds, struct timeval \*timeout)* - системныйвызов, показывающий, какиедескрипторыиззаданныхнаборовготовыдлячтения, записиилиожидаютобработкуошибок.

*struct timeval*

*{*

*long tv\_sec;*

*long tv\_usec;*

*}*

*void FD\_ZERO(fd\_set \*fdset); // инициализациябитовоймаскинулями*

*void FD\_SET(int fd, fd\_set \*fdset) // установкабита fd ( вединицу )*

*int FD\_ISSET(int fd,fd\_set \*fdset) // проверкабита fd*

*void FD\_CLR(int fd,fd\_set \*fdseе) // сбитьбит fd ( нулём )*

Первый параметр – число дескрипторов файлов, которые могут представлять интерес для процесса. Можно использовать FD\_SETSIZE ( максимальное число дескрипторов файлов, которые могут быть переданы вызовом select ).По умолчанию уже существует три дескриптора – stdin, stdout, stderr. Тип fd\_set используется для определения размера переменной для хранения битового вектора ( маски ), в которой каждый бит соответствует дескриптору файла ( 1 – да, 0 – нет ).

Если structtimeval равен нулю, то select немедленно завершится, иначе при не активных файловых дескрипторах возврат из вызова произойдёт через указанное время.

Функция возвращает -1 при ошибке, 0 при истечении интервала ожидания, или число дескрипторов, обработанных процессом.

*Пример:*

*/\* взаимодействие родительского процесса с 3 сыновьями через каналы \*/*

*#include <sys/time.h>*

*#include <sys/wait.h>*

*#defin M\_S 7*

*char \*msg1=”Привет”;*

*char \*msg2=”Пока!!”;*

*void parent(int p[3][2])*

*{ char buf[M\_S],ch;*

*fd\_set set,sset;*

*int i;*

*for (i=0;i<3;i++) close(p[i][1]);*

*FD\_ZERO(&sset);*

*FD\_SET(0,&sset);*

*for (i=0;i<3;i++) FD\_SET(p[i][0],&sset);*

*while (set=sset,select(p[2][0]+1,&set,NULL,NULL,NULL)>0)*

*{*

*if (FD\_ISSET(0,&set)) printf(“Изстандартноговвода\n”);*

*read(0,&ch,1);*

*printf(“%c\n”,ch);*

*for (i=0;i<3;i++)*

*if (FD\_ISSET(p[i][0],&set))*

*if read(p[i][0].buf,M\_S]>0)*

*printf(Сообщение от сына %d\nMSG=&s\n”,I,buf);*

*if (waitpid(-1,NULL,WNOHANG)==-1) return;*

*}*

*}*

*int child(int p[2])*

*{ int i;*

*close(p[0]);*

*for (i=0;i<2;i++)*

*{*

*write(p[1],msg1,M\_S);*

*sleep(1);*

*}*

*write(p[1],msg2,M\_S);*

*exit(0);*

*}*

*int main()*

*{ int pip[3][2],i;*

*for (i=0;i<3;i++)*

*{*

*pipe(pip[i]);*

*switch (fork())*

*{*

*case -1 : printf(“Ошибка fork\n”); exit(1);*

*case 0 : child(pip[i]);*

*}*

*}*

*parent(pip);*

*exit(0)*

*}*

С помощью именованых каналов можно устанавливать связь не только с родственными процессами.

*Прототип:*

*#include <sys/types.h>*

*#include <sys/stat.h>*

*int mkfifol (const char \*pathname, mode\_t mode);*

*Пример использования:*

*intfd;*

*mkfifo(“/tmp/fifo1”,0666);*

1. *fd=open(“/tmp/fifo1”,O\_WRONLY);*
2. *if ((fd=open(“/tmp/fifo1”,O\_WRONLY|O\_NONBLOCK))==-1)*

*printf(“Ошибка open для канала FIFO\n”);*

Данные команды будут по-разному себя вести. В первом случае процесс ожидает, пока другой процесс не откроет канал для противоположного действия. Во втором случае в errno::=ENXIO.

*Пример:*

*/\* использование именнованых каналов \*/*

*#include <fcntl.h>*

*#include <stdio.h>*

*#include <errno.h>*

*#define M\_S 64*

*char \*fifo=”FIFO\_1”*

*main(int argc, char \*argv[])*

*{ int fd,j,n\_w;*

*char ms\_buf[M\_S];*

*if (argc<2) { printf(“Нет передаваемого сообщения”\n”); exit(1); }*

*if ((fd=open(fifo,O\_WRONLY|O\_NONBLOCK))<0)*

*{ printf(“Ошибкавызова open для fifo”); exit(2); };*

*for (j=1;j<argc;j++)*

*{*

*if (strlen(argv[j])>M\_S-1)*

*{ printf(“Шлишком длинное сообщение %s\n”,argv[j]); continue; };*

*strcpy(ms\_buf,argv[j]);*

*if ((n\_w=write(fd,ms\_buf,M\_S))==-1)*

*{ printf(“Ошибка записи сообщения\n”); exit(3); };*

*}*

*exit(0);*

*}*

*- - - -*

*main(int argc< char \*argv[])*

*{ int fd,i;*

*char ms\_buf[M\_S];*

*if (mrfifo(fifo,0666)==-1)*

*{ printf(“Ошибкаприоткрытии fifo\n”); exit(2); }*

*for (i=0;i<20;i++)*

*{*

*if (read(fd,ms\_buf,M\_S)<0)*

*{ printf(“Ошибка чтения из канала\n”); exit(3); }*

*printf(“Получено сообщение:%s\n”,ms\_buf);*

*}*

*exit(0);*

*}*

**Взаимодействие процессов с использованием сигналов**

1. Сигналы являются средством передачи уведомления о некотором произошедшем событии между процессами или между ядром системы и процессами. Сигналы рассматриваются как форма межпроцессорного взаимодействия, хотя по сути напоминают прерывания, генерируемые при нарушении нормального выполнения процесса. Каждый сигнал имеет уникальное имя и имеются два системных вызова для генерации сигналов.

*Прототипы:*

*#include <sys/types.h>*

*#include <signal.h>*

*int kill (pid\_t pid, int sig);*

*int raise (int sig);*

Системный вызов **kill** посылает сигнал процессу или группе процессов. Тип сигнала определяется аргументом **sig**, а первый аргумент определяет, кому посылается сигнал.

Если *PID>0*, сигнал посылается процессу, идентификатор которого равен PID.

Если *PID=0*, то сигнал посылается всем процессам из группы процессов, к которой принадлежит процесс, посылающий сигнал.

Если *PID<1*, то сигнал посылается всем процессам, идентификатор группы которых равен абсолютному значению pid.

Второй системный вызов **raise** предназначается для генерации сигнала **sig** для текущего процесса.

Системные вызовы возвращают 0 в случае успешной посылки сигнала. **Kill** в случае ошибки возвращает –1, а **raise** – ненулевое значение.

В случае ошибки обнаруживаются ошибочные ситуации:

*EINVAL* - сигнал не является номером сигнала;

*EPERM* - процесс не правомочен посылать сигнал указанному принимающему процессу;

*ESRCH* - pid не является идентификатором процесса или группы процессов.

К генерации сигнала могут привести следующие ситуации:

1. Ядро посылает процессу или группе процессов сигнал при нажатии пользователем определенных клавиш или их комбинаций (например, Ctrl-C).

2. Аппаратные особые ситуации (деление на 0, обращение к недопустимой области памяти, нарушение защиты памяти, отсутствие требуемой страницы в памяти и т.д.). Обычно эти ситуации определяются аппаратурой машины, и ядру операционной системы посылаются соответствующие уведомления, например, в виде прерывания, а ядро реагирует на такое событие отправкой соответствующего сигнала процессу, который находился в стадии выполнения, когда произошла особая ситуация.

3. Определенные программные состояния системы или ее компонентов. Эти причины имеют чисто программный характер. К такой ситуации относится посылка сигнала по срабатыванию таймера, установленного с помощью специального системного вызова **alarm**.

При получении сигнала может быть выбрано одно из следующих действий реагирования:

- игнорирование сигнала;

- перехват и самостоятельная обработка сигнала;

- выполнение действия по умолчанию.

Есть два сигнала (SIGKILL и SIGSTOP), которые невозможно ни игнорировать, ни перехватывать.

Существует определенный набор сигналов. В качестве действия по умолчанию при получении сигнала могут быть:

- завершение текущего процесса;

- завершение и создание файла **core**;

- игнорирование.

Таблица 1 - Основные сигналы ОС UNIX

|  |  |  |  |
| --- | --- | --- | --- |
|  | Название | Действие по умолчанию | Комментарии |
| 1 | SIGABRT | Завершить +core | Сигнал отправляется, если процесс вызывает системный вызов **abort**. |
| 2 | SIGALRM | Завершить | Сигнал отправляется, когда срабатывает таймер, ранее установленный с помощью системного вызова **alarm**. |
| 3 | SIGBUS | Завершить +core | Сигнал свидетельствует о некоторой аппаратной ошибке. Обычно отправляется при обращении к допустимому виртуальному адресу, для которого отсутствует соответствующая физическая страница. |
| 4 | SIGCHLD | Игнорирование | Сигнал, посылаемый родительскому процессу при завершении выполнения его потомка. |
| 5 | SIGEVG | Завершить +core | Сигнал свидетельствует о попытке обращения к недопустимому адресу или к области памяти, для которой у процесса недостаточно привилегий. |
| 6 | SIGFPE | Завершить +core | Сигнал свидетельствует о возникновении особых ситуаций, таких, как деление на ноль или переполнение операций с плавающей точкой. |
| 7 | SIGHUP | Завершить | Сигнал посылается лидеру сеанса, связанному с управляющим терминалом, когда ядро обнаруживает, что терминал отсоединился. Сигнал также посылается всем процессам группы при завершении работы лидера. Этот сигнал может быть использован при необходимости передачи сообщений процессам-демонам. |
| 8 | SIGILL | Завершить +core | Сигнал посылается ядром, если процесс попытался выполнить недопустимую операцию. |
| 9 | SIGINT | Завершить | Сигнал посылается всем процессам данной группы при нажатии клавиш Ctrl-C или Del. |
| 10 | SIGKILL | Завершить | Сигнал, при получении которого выполнение процесса завершается. Его нельзя ни перехватывать, ни игнорировать. |
| 11 | SIGPIPE | Завершить | Сигнал посылается при попытке записи в канал, получатель данных которого закрыл дескриптор (завершил свое выполнение). |
| 12 | SIGPOLL | Завершить | Сигнал отправляется при наступлении определенного события для устройства, которое является опрашиваемым. |
| 13 | SIGPWR | Игнорирование | Сигнал генерируется при угрозе потери питания. Обычно отправляется, когда питание системы переключается на источник бесперебойного питания. |
| 14 | SIGQUIT | Завершить +core | Сигнал посылается ядром всем процессам группы при нажатии клавиш Ctrl-\. |
| 15 | SIGSTOP | Остановка | Сигнал отправляется всем процессам группы при нажатии клавиш Ctrl-Z. Получение сигнала вызывает остановку выполнения процесса. |
| 16 | SIGSYS | Завершить +core | Сигнал отправляется ядром при попытке недопустимого системного вызова. |
| 17 | SIGTERM | Завершить | Сигнал представляет собой предупреждение о том, что процесс вскоре будет уничтожен, что позволяет процессу подготовиться к своему завершению. **Kill** (1) посылает такой сигнал. |
| 18 | SIGTTIN | Остановка | Сигнал генерируется ядром при попытке процесса фоновой группы осуществить чтение с управляющего терминала. |
| 19 | SIGTTOU | Остановка | Сигнал генерируется ядром при попытке процессов фоновой группы осуществить запись на управляющий терминал. |
| 20 | SIGUSR1 | Завершить | Сигнал, предназначенный для прикладных задач, как средство межпроцессорного взаимодействия. |
| 21 | SIGUSR2 | Завершить | Сигнал, предназначенный для прикладных задач, как средство межпроцессорного взаимодействия. |

Для определения собственных обработчиков сигналов имеются специальные системные вызовы. Самый простейший такой системный вызов signal.

*Прототип:*

*#include<signal.h>*

*void (\* signal (int sig, void (\* disp)(int)))(int);*

Простейшим интерфейсом к сигналам является системная функция signal, которая позволяет изменить предопределенную реакцию на сигнал, определенную ОС UNIX. Порожденный вызовом fork процесс наследует установленную реакцию на сигнал от своего отца. Но при выполнении exec эта реакция устанавливается по умолчанию.

Применяя системный вызов signal процесс имеются слабые возможности управления сигналами:

1. Процесс не может заблокировать сигнал, т.е. отложить получение сигнала на период выполнения критического участка кода.

2. Каждый раз при получении сигнала реакция на этот сигнал устанавливается на действие по умолчанию.

*Пример:*

*/\* установка действий на реагирование сигналов \*/*

*static void sig\_hndlr (int sig\_no) //функцияобработкисигнала*

*{*

*signal (SIGINT,sig\_hndlr);*

*printf (“Полученсигнал SIGINT\n”);*

*}*

*main( )*

*{*

*signal (SIGINT,sig\_hndlr);*

*signal (SIGUSR1,SIG\_DFL);//установка на действие по умолчанию системы*

*signal (SIGUSR2,SIG\_IGN); // установка на игнорирование сигнала*

*while (1) pause ( ); // остновка программы до получения сигнала*

*}*

Модель сигналов в POSIX основана на понятии набора сигналов, описываемого переменной типа sigset\_t. Каждый бит этой переменной отвечает за один сигнал. Во многих системах этот тип имеет длину 32 бита, т.е. имеется возможность использовать 32 сигнала.

Для управления наборами сигналов применяются следующие функции:

*Прототипы:*

*#include <signal.h>*

*int sigemptyset (sigset\_t \*set);*

*int sigfillset (sigset\_t \*set);*

*int sigaddset (sigset\_t \*set, int signo);*

*int sigdelset (sigset\_t \*set, int signo);*

*int sigismember (sigset\_t \*set, int signo);*

1. Функция **sigemptyset** инициализирует набор, очищая все биты.

2.Функция **sigfillset** устанавливает в набор все сигналы, известные системе.

3. Функции **sigaddset** добавляет сигнал в набор.

4. Функции **sigdelset** удаляет сигнал из набора.

5. Функция **sigismember** проверяет, входит ли сигнал signo в набор.

Стандартом POSIX также определён системный вызов

*int sigaction(int sig struct sigaction \*act, struct sigaction \*oact);*

который позволяет установить диспозицию сигнала, узнать её текущее значение или выполнить то и другое одновременно. В этой функции вся необходимая информация передаётся через указатель на структуру типа sigaction, которая имеет следующие поля:

*struct sigaction*

*{*

*void (\*sa\_handler)();*

*void (\*sa\_sigaction)(int, siginfo\_t \*, void \*);*

*sigset\_t sa\_mask;*

*int sa\_flags;*

*}*

Первое поле структуры определяет обработчик сигналов, второе поле - обработка сигналов, если установлен флаг SA\_SIGINFO, третье поле - маска сигналов, четвёртое поле - поле флагов.

Поле **sa\_handler**⎯ определяет действие, которое необходимо выполнить при получении сигналов. Может содержать значения:

SIG\_IGN - игнорировать сигнал;

SIG\_DFL - обратботчик по умолчанию;

<АДРЕС> - адрес функции обработчика.

Если поля **sa\_handler** и **sa\_sigaction** не равны NULL то **sa\_mask** содержит набор сигналов, которые будут добавлены к маске сигналов перед вызовом обработчика.

Каждый процесс имеет установленную маску сигналов, определяющая сигналы, доставка которых должна быть заблокирована. Если бит маски установлен, то соответствующий ему сигнал будет заблокирован. После возврата из функции обработчика, значение маски возвращается в исходное состояние. Сигнал, для которого установлен обработчик, также блокируется перед вызовом функции обработки.

Поле **sa\_flags** определяет флаги, которые определяют способ доставки сигнала:

*SA\_ONSTACK*⎯ если определена функция-обработчик сигнала и с помощью функции **sigaltstack()** задан альтернативный стек для функции-обработчика, то при обработке сигнала будет использоваться этот стек. Если флаг не установлен, будет использоваться обычный стек процесса;

*SA\_RESETHAND*⎯ если определена функция-обработчик, то реакция на сигнал будет изменена на SIG\_DFL и сигнал не будет блокироваться при запуске обработчика. Если флаг не установлен, то реакция на сигнал не изменяется.

*SA\_NODEFERR*⎯ если определена функция-обработчик, то сигнал блокируется на время обработки только в том случае, когда он явно указан в поле **sa\_mask**. Если флаг не установлен, то в процессе обработки данный сигнал автоматически блокируется в процессе обработки.

*SA\_RESTART*⎯ если определена функция-обработчик, то ряд системных вызовов, выполнение которых было прервано полученным сигналом, будет автоматически запущен после обработки сигнала. К таким системным вызовам относятся: write, read. Если флаг не установлен, то системный вызов возвращает ошибку EINTR (ошибка прерывания).

* *SA\_SIGINFO*⎯ если указана реакция на перехват сигнала, то вызывается функция обработчик определенная **sa\_sigaction**. Если флаг не установлен, то вызывается функция обработчик определенная полем **sa\_handler**.
* *SA\_NOCHLDWAIT*⎯ если указанный аргументом **sig** сигнал равен SIGCHLD, то при завершении потомки не будут переходить в состояние “зомби”. Если процесс в дальнейшем вызовет одну из функций семейства wait, их выполнение будет блокировано до завершения работы всех потомков данного процесса.
* *SA\_NOCLDSTOP*⎯ если sig равен SIGCHLD, то указанный сигнал не будет отправляться процессу при завершении или остановке любого из его потомков.

Если установлен SA\_SIGINFO, то при получении сигнала будет вызван обработчик, адресованный sa\_sigaction. Обработчику передаётся номер сигнала и указатель на структуру типа siginfo\_t, содержащую информацию о причинах получения сигнала, а также указатель на структуру типа ucontext\_t, содержащую контекст процесса. Эта

Структура siginfo\_t включает следующие поля:

intsi\_signo - номер сигнала;

int si\_errno - номер ошибки;

int si\_code - причина отправления сигнала.

Если **si\_code** меньше или равно нулю, то сигнал был отправлен прикладным процессом и структура siginfo\_tсодержит поля **:**

pid\_t si\_id - идентификатор процесса PID;

uid\_t si\_uid - идентификатор пользователя UID,

которые адресуют процесс, пославший сигнал. Если значение **si\_code** болеше нуля, то оно указывает на причину отправления сигнала.

Системный вызов **sigprocmask** позволяет получить и установить текущую маску сигналов:

*int sigprocmask (int how, sigset\_t \*set, sigset\_t \*oset).*

Параметр how может принимать значения:

SIG\_BLOCK - результирующая маска получается путем объединения текущей маски сигналов и набора сигналов set;

SIG\_UNBLOCK - результирующая маска получается путем удаления набора сигналов set из текущей маски;

SIG\_SETMASK -маска будет заменена на набор сигналов set.

Если набор set есть NULL, то первый аргумент игнорируется, а если oset не равен NULL, то по указанному адресу помещается текущая маска сигналов.

*Прототипы:*

*int sigpending (sigset\_t \*set).*

*int sigsuspend (const sigset\_t \*set );*

Системный вызов **sigpending** используется для получения набора заблокированных сигналов и сигналов, ждущих передачи. Системный вызов **sigsuspend** замещает текущую маску набором **set** и приостанавливает выполнение процесса до получения сигналов, диспозиция которых установлена, либо на завершение процесса, либо на вызов функции-обработчика/

*Примеры:*

*/\* - перехват и самостоятельная обработка сигнала SIGINT \*/*

*#include <signal.h>*

*void catch\_int(int sigmo)*

*{*

*printf(“\nСигнал signo=%d\n”,signo);*

*printf(“Возврат из обработчика”);*

*}*

*main()*

*{ static struct sigaction act;*

*act.sa\_handler=catch\_int;*

*sigfillset(&act.sa\_mask);*

*sigaction(SIGINT, &act,NULL);*

*printf(“Вызов sleep(1)\n”);*

*sleep(1);*

*…*

*printf(“Завершение программы\n”);*

*exit(0);*

*}*

*/\* установка игнорирования на сигнал SIGTERM*

*с последующим возвратом к старому обработчику \*/*

*#include <signal.h>*

*main()*

*{ static struct sigaction act,old\_ act;*

*sigaction(SIGTERM, NULL,&old\_act);*

*act.sa\_handler=SIG\_IGN;*

*sigaction(SIGTERM, &act,NULL);*

*…*

*sigaction(SIGTERM, &old\_act,NULL);*

*exit(0);*

*}*

*#include <signal.h>*

*#include <sys/types.h>*

*#include <sys/stat.h>*

*#include <fcntl.h>*

*#include <unistd.h>*

*int (\*mysignal) (int sig\_no, void (\*hndlr)(int) )*

*{*

*struct sigaction act, oact;*

*act.sa\_handler=hndlr;*

*sigemptyset(&act.sa\_mask);*

*act.sa\_flags=0;*

*if (sig\_no != SIGALARM) act.sa\_flags |= SA\_RESTART;*

*if (sigaction(sig\_no, &act, &oact) < 0) return (SIG\_ERR);*

*return (oact.sa\_handler);*

*}*

*static void sig\_hndlr(int sig\_no)*

*{*

*printf(“Полученсигнал %i \n”,sig\_no);*

*}*

*main()*

*{*

*mysignal(SIGINT, sig\_hndlr);*

*mysignal(SIGUSR1, SIG\_DFL);*

*mysignal(SIGUSR2, SIG\_IGN);*

*while(1)*

*pause();*

*}*

*/\* взаимодействие родственных процессов ( отец и сын )*

*спомощьюсигнала SIGUSR1 \*/*

*#include <signal.h>*

*int ntimes=0;*

*void p\_action(int sig)*

*{ printf(“Родитель получил сигнал %d\n”,++ntimes); }*

*void c\_actiom(int sig)*

*{ printf(“Сынполучилсигнал %d\n”,++ntimes); }*

*main()*

*{*

*pid\_t pid, ppid;*

*static sig\_action p\_act,c\_act;*

*p\_act.sa\_handler=p\_action;*

*sigaction(SIGUSR1,&p\_act,NULL);*

*switch (pid=fork())*

*{*

*case 0 : c\_act.sa\_handler=c\_action;*

*sigaction(SIGUSR1,&c\_act,NULL);*

*ppid=getppid();*

*for(;;)*

*{*

*sleep(10);*

*kill(ppid,SIGUSR1);*

*pause();*

*}*

*break;*

*default: for (;;)*

*{*

*pause();*

*sleep(1);*

*kill(pid,SIGUSR1);*

*}*

*}*

*}*

**Ограничения для процессов**

Так как ОС UNIX является многозадачной системой, то в ходе работы несколько процессов могут конкурировать между собой за доступ к различным ресурсам. Для справедливого распределения ресурсов (память, дисковое пространство) каждому из процессов устанавливается индивидуальный набор ограничений.

Для получения информации о текущих ограничениях используются системные вызовы:

*#include <sys/times.h>*

*#include <sys/resource.h>*

*int getrlimit (int resource, struct rlimit \*rlp);*

*int setrlimit (int resource, struct rlimit \*rlp);*

*struct rlimit*

*{*

*rlim\_t rlim\_cur;*

*rlim\_t rlim\_max;*

*}*

Параметр **resource** определяет вид ресурса, для которого нужно узнать или изменить ограничения. Параметр **rlim\_cur** определяет изменяемое ограничение, т.е. текущее ограничение процесса на данный ресурс. Параметр **rlim\_max** определяет жесткое ограничение, т.е. максимально возможное значение для данного ресурса.

Любой процесс может изменить текущее значение ограничения ресурса до максимально возможного значения. Жесткое ограничение может быть изменено в сторону увеличения только процессом с привилегиями суперпользователя. Обычные процессы могут только уменьшать его. Обычно ограничения устанавливаются при инициализации системы и затем наследуются порожденными процессами, хотя они могут изменяться и потом. Максимально возможный предел потребления ресурса может иметь значение, определяемое физическим ограничением системы. В этом случае в поле **rlim\_max** должно быть установлено значение RLIM\_INFINITY.

В системе определены следующие ограничения:

*RLIMIT\_CORE* - максимальный размер создаваемого файла **core**, содержащего в памяти образ процесса. Если значение установлен в 0, то файл создаваться не будет.

*RLIMIT\_CPU* - максимальное время использования процессора в секундах. При превышении промежутка времени процессу посылается сигнал SIGXCPU.

*RLIMIT\_DATA* - максимальный размер сегмента данных процесса в байтах. При достижении этого предела последующие вызовы функций распределения памяти завершаются ошибкой ENOMEM.

*RLIMIT\_FSIZE* - максимальный размер файла, который может создать процесс. Если установить этот параметр в 0, то процесс не может создавать файлы. При достижении файлом заданного предела посылается сигнал SIGXFSZ.

*RLIMIT\_NOFILE* - максимальное количество назначенных файловых дескрипторов процесса (т.е. максимальное число файлов, которые могут быть открыты процессом одновременно). Если процесс попытается получить больше дескрипторов, чем задано, то ему возвращается ошибка EMFILE.

*RLIMIT\_STACK* - максимальный размер стека процесса. При попытке выхода за предел процессу отправляется сигнал SIGSEGV.

*RLIMIT\_VMEM* - максимальный размер отображаемой памяти процесса в байтах. При превышении этого предела, использование функций распределения памяти возвращают ошибку ENOMEM.

*RLIMIT\_NPROC* - максимальное число процессов с одним реальным UID. Если достигнут предел, то вызов **fork** завершится с ошибкой EAGAIN.

*RLIMIT\_RSS* - максимальный размер в байтах резидентной части процесса. Определяет максимальный размер выделяемый процессу физической памяти. Если система ощущает нехватку оперативной памяти, то она освободит память за счет процессов превышавших свой ресурс.

*RLIMIT\_MEMLOCK* - максимальный размер в байтах физической памяти, которую процесс может заблокировать с помощью системного вызова mallock. При превышении размера mallock завершается с ошибкой EAGAIN.

*Пример:*

*/\* получение ограничения для процесса \*/*

*#include <stdio.h>*

*#include <sys/time.h>*

*#include <sys/resource.h>*

*void disp\_limit(int resource, char \*r\_name)*

*{ struct rlimit rlm;*

*getrlimit(resource, &rlm);*

*printf("%-13s", r\_name);*

*if (rlm.rlim\_cur == RLIM\_INFINITY) printf("INFINITY \n");*

*else printf("\n%10ld ", rlm.rlim\_cur);*

*if (rlm.rlim\_max == RLIM\_INFINITY) printf("INFINITY \n");*

*else printf("\n%10ld ", rlm.rlim\_max);*

*}*

*main(void)*

*{*

*disp\_limit(RLIMIT\_NOFILE,"RLIMIT\_CORE");*

*disp\_limit(RLIMIT\_NOFILE,"RLIMIT\_FSIZE");*

*}*

**Дополнительные средства межпроцессного взаимодействия**

.1 Введение и основные понятия

ОС UNIX предлагает множество дополнительных механизмов межпроцессного взаимодействия. Их наличие дает UNIX богатые возможности в области связи между процессами и позволяет разработчику использовать различные подходы при программировании многозадачных систем. Дополнительные средства межпроцессного взаимодействия, которые будут рассмотрены, можно разбить на следующие категории:

- передача сообщений;

-семафоры;

-разделяемая память.

Эти средства широко применяются и ведут свое начало от системы UNIXSystemV, поэтому их иногда называют *IPCSystemV*. Следует заметить, что вышеназванные дополнительные средства были определены в последних версиях стандарта POSIX.[[1]](#footnote-1)

**Ключи средств межпроцессного взаимодействия**

Программный интерфейс всех трех средств *IPCSystem V* однороден, что отра­жает схожесть их реализации в ядре. Наиболее важным из общих свойств являет­ся *ключ* (key). Ключи - это числа, обозначающие объект межпроцессного взаи­модействия в системе UNIX примерно так же, как имя файла обозначает файл. Другими словами, ключ позволяет ресурсу межпроцессного взаимодействия со­вместно использоваться несколькими процессами. Обозначаемый ключом объект может быть очередью сообщения, набором семафоров или сегментом разделяемой памяти. Ключ имеет тип *key\_t*, состав которого зависит от реализации и опреде­ляется в системном заголовочном файле *<sys/types .h>.*

Ключи не являются именами файлов и несут меньший смысл. Они должны выбираться осторожно во избежание конфликта между различными программами, в этом помогает применение дополнительной опции – «версии проекта». В ОС UNIX существует простая библиотечная функция *ftok*, которая образует ключ по указанному файлу.

**Описание**

*#include <sys/ipc.h>*

*key\_t ftok(const char \*path, int id);*

Эта процедура возвращает номер ключа на основе информации, связанной с файлом *path*. Параметр *id* также учитывается и обеспечивает еще один уровень уникальности - «версию проекта»; другими словами, для одного имени *path*бу­дут получены разные ключи при разных значениях *id*. Процедура *ftok*не слиш­ком удобна: например, если удалить файл, а затем создать другой с таким же име­нем, то возвращаемый после этого ключ будет другим. Она завершится неудачей и вернет значение (key\_t) -1 и в случае, если файл *path*не существует. Проце­дуру *ftok*можно применять в приложениях, использующих функции межпро­цессного взаимодействия для работы с определенными файлами или при приме­нении для генерации ключа файла, являющегося постоянной и неотъемлемой частью приложения.

**Операция *get***

Программа применяет ключ для создания объекта межпроцессного взаимодей­ствия или получения доступа к существующему объекту. Обе операции вызыва­ются при помощи операции *get*. Результатом операции *get*является его цело­численный *идентификатор* (facilityidentifier), который может использоваться при вызовах других процедур межпроцессного взаимодействия. Если продолжить аналогию с именами файлов, то операция *get*похожа на вызов *creat*или *open*, а идентификатор средства межпроцессного взаимодействия ведет себя подобно дескриптору файла. В действительности, в отличие от дескрипторов файла, иден­тификатор средства межпроцессного взаимодействия является уникальным. Раз­личные процессы будут использовать одно и то же значение идентификатора для объекта межпроцессного взаимодействия.

В качестве примера рассмотрим вызов *msgget*для создания новой очереди со­общений (что представляет собой очередь сообщений, обсудим позже):

*mqid = msgget( (key\_t)0100, 0644 | IPC\_CREAT | IPC\_EXCL);*

Первый аргумент вызова, *msgget*, является ключом очереди сообщений. В слу­чае успеха процедура вернет неотрицательное значение в переменной mqid, кото­рая служит идентификатором очереди сообщений. Соответствующие вызовы для семафоров и объектов разделяемой памяти называются соответственно *semget* и *shmget*.

**Структуры данных статуса**

При создании объекта межпроцессного взаимодействия система также созда­ет *структуру статуса средства межпроцессного взаимодействия*(IPCfacilitystatusstructure), содержащую всю управляющую информацию, связанную с объектом. Для сообщений, семафоров и разделяемой памяти существуют разные типы структуры статуса. Каждый тип содержит информацию, свойственную этому средству межпроцессного взаимодействия. Тем не менее все три типа структу­ры статуса содержат общую структуру прав доступа. Структура прав доступа *ipc\_perm*содержит следующие элементы:

*uid\_tcuid; /\* Идентификатор пользователя создателя объекта \*/*

*gid\_tcgid; /\* Идентификатор группы создателя объекта \*/*

*uid\_tuid; /\* Действующий идентификатор пользователя \*/*

*gid\_tgid; /\* Действующий идентификатор группы \*/*

*mode\_tumode; /\* Права доступа \*/*

Права доступа определяют, может ли пользователь выполнять «чтение» из объекта (получать информацию о нем) или «запись» в объект (работать с ним). Коды прав доступа образуются точно таким же образом, как и для файлов. Поэто­му значение 0644 для элемента *umode*означает, что владелец может выполнять чтение и запись объекта, а другие пользователи - только чтение из него. Обратите внимание, что права доступа, заданные элементом *umode*, применяются в сочета­нии с действующими идентификаторами пользователя и группы (записанными в элементах *uid*и *gid*).[[2]](#footnote-2)Очевидно также, что права на выполнение в данном случае не имеют значения. Как обычно, суперпользователь имеет неограниченные полномочия. В отличие от других конструкций UNIX, значение переменной *umask*пользователя не действует при создании средства межпроцессного взаимодействия.

.2. Очереди сообщений

В сущности, сообщение является просто последовательностью символов или байтов (необязательно заканчивающейся нулевым символом). Сообщения переда­ются между процессами при помощи *очередей сообщений*(messagequeues), которые можно создавать или получать к ним доступ при помощи вызова *msgget*. После создания очереди процесс может помещать в нее сообщения при помощи вызова *msgsnd*, если он имеет соответствующие права доступа. Затем другой процесс может считать это сообщение при помощи примитива *msgrcv*, который извлекает сообще­ние из очереди. Таким образом, обработка сообщений аналогична обмену данными при помощи вызовов чтения и записи для каналов (рассмотренном в разделе ).

Функция *msgget* определяется следующим образом:

**Описание**

*#include <sys/msg.h>*

*int msgget (key\_t key, int permflags);*

Этот вызов лучше всего представить как аналог вызова *open* или *creat*. Параметр *key,* который, в сущности, является простым числом, идентифицирует очередь сообщений в системе. В случае успеш­ного вызова, после создания новой очереди или доступа к уже существующей, вызов *msgget* вернет ненулевое целое значение, которое называется *идентифика­тором очереди сообщений*(messagequeueidentifier).

Параметр *permflags* указывает выполняемое вызовом *msgget* действие, которое задается при помощи двух констант, определенных в файле <sys/ipc.h>; они могут использоваться по отдельности или объединяться при помощи операции побитового ИЛИ:

IPC\_CREATПри задании этого флага вызов *msgget* создает новую очередь сообщений для данного значения, если она еще не существует. Если продолжить аналогию с файлами, то при задании этого флага вызов *msgget* выполняется в соответствии с вызовом *creat*, хотя очередь сообщений и не будет «перезаписана», если она уже существует. Если же флаг IPC\_CREAT не установ­лен и очередь с этим ключом существует, то вызов *msgget* вер­нет идентификатор существующей очереди сообщений

IPC\_EXCLЕсли установлен этот флаг и флаг IPC\_CREAT, то вызов пред­назначается только для создания очереди сообщений. Поэто­му, если очередь с ключом *key* уже существует, то вызов *msgget* завершится неудачей и вернет значение -1. Переменная *errno* будет при этом содержать значение EEXIST.

При создании очереди сообщений младшие девять бит переменной *permflags* используются для задания прав доступа к очереди сообщений аналогично коду доступа к файлу. Они хранятся в структуре *ipc\_perm*, создаваемой одновременно с самой очередью.

Теперь можно вернуться к примеру.

*mqid = msgget( (key\_t)0100, 0644 | IPC\_CREAT | IPC\_EXCL);*

Этот вызов предназначен для создания (и только создания) очереди сообще­ний для значения ключа равного (key\_t) 0100. В случае успешного завершения вызова очередь будет иметь код доступа 0644. Этот код интерпретируется таким же образом, как и код доступа к файлу, обозначая, что создатель очереди может отправлять и принимать сообщения, а члены его группы и все остальные могут выполнять только чтение. При необходимости для изменения прав доступа или владельца очереди может использоваться вызов *msgctl*.

**Работа с очередью сообщений: примитивы *msgsnd*и *msgrcv***

После создания очереди сообщений для работы с ней могут использоваться два следующих примитива:

**Описание**

*#include <sys/msg.h>*

*int msgsnd(int mqid, const void \*message, size\_t size, int flags);*

*int msgrcv(int mqid, void \*message, size\_t size, long msg\_type, int flags);*

Первый из вызовов, *msgsnd*, используется для добавления сообщения в оче­редь, обозначенную идентификатором *mqid*.

Сообщение содержится в структуре *message*- шаблоне, определенном пользо­вателем и имеющем следующую форму:

*struct mymsg*

*{*

*long mtype; /\* Типсообщения \*/*

*char mtext[SOMEVALUE]; /\* Текстсообщения \*/*

*};*

Значение поля *mtype*может использоваться программистом для разбиения сообщений на категории. При этом значимыми являются только положительные значения; отрицательные или нулевые не могут использоваться (это будет видно из дальнейшего описания операций передачи сообщений). Массив *mtext*служит для хранения данных сообщения (постоянная *SOMEVALUE*выбрана совершенно произвольно). Длина посылаемого сообщения задается параметром *size*вызова *msgsnd*и может быть в диапазоне от нуля до меньшего из двух значений *SOMEVALUE*и максимального размера сообщения, определенного в системе.

Параметр *flsgs*вызова *msgsnd*может нести только один флаг: *IPC\_NOWAIT*. При неустановленном параметре *IPC\_NOWAIT*вызывающий процесс приостано­вит работу, если для посылки сообщения недостаточно системных ресурсов. На практике это произойдет, если полная длина сообщений в очереди превысит мак­симум, заданный для очереди или всей системы. Если флаг *IPC\_NOWAIT*установ­лен, тогда при невозможности послать сообщение возврат из вызова произойдет немедленно. Возвращаемое значение будет равно -1, и переменная *errno*будет иметь значение *EAGAIN*, означающее необходимость повторения попытки.

Вызов *msgsend*также может завершиться неудачей из-за установленных прав доступа. Например, если ни действующий идентификатор пользователя, ни дей­ствующий идентификатор группы процесса не связаны с очередью, и установлен код доступа к очереди 0660, то вызов *msgsnd*для этой очереди завершится неуда­чей. Переменная *errno*получит значение *EACCES*.

Для чтения из очереди, заданной идентификатором *mqid*, используется вызов *msgrcv*. Чтение разрешено, если процесс имеет права доступа к очереди на чтение. Успешное чтение сообщения приводит к удалению его из очереди.

На этот раз переменная *message*используется для хранения полученного со­общения, а параметр *size* задает максимальную длину сообщений, которые мо­гут находиться в этой структуре. Успешный вызов возвращает длину полученно­го сообщения.

Параметр *msg\_type* определяет тип принимаемого сообщения, он помогает вы­брать нужное из находящихся в очереди сообщений. Если параметр *msg\_type* равен нулю, из очереди считывается первое сообщение, то есть то, которое было послано первым. При ненулевом положительном значении параметра *msg\_type* считывает­ся первое сообщение из очереди с заданным типом сообщения. Например, если оче­редь содержит сообщения со значениями *mtype* 999, 5 и 1, а параметр *msg\_type* в вызове *msgrcv* имеет значение 5, то считывается сообщение типа 5. И, наконец, если параметр *msg\_type* имеет ненулевое отрицательное значение, то считывается первое сообщение с наименьшим значением *mtype*, которое меньше или равно мо­дулю параметра *msg\_type*. Этот алгоритм кажется сложным, но выражает простое правило: если вернуться к нашему предыдущему примеру с тремя сообщениями со значениями *mtype* 999, 5 и 1, то при значении параметра *msg\_type* в вызове *msgrcv* равном -999 и троекратном вызове сообщения будут получены в порядке 1, 5, 999.

Последний параметр *flags* содержит управляющую информацию. В этом пара­метре могут быть независимо установлены два флага - *IPS\_NOWAIT*и *MSG*\_*NOERROR*. Флаг *IPC*\_*NOWAIT*имеет обычный смысл - если он не задан, то процесс будет при­остановлен при отсутствии в очереди подходящих сообщений, и возврат из вызо­ва произойдет после поступления сообщения соответствующего типа. Если же этот флаг установлен, то возврат из вызова при любых обстоятельствах произой­дет немедленно.

При установленном флаге *MSG*\_*NOERROR*сообщение будет усечено, если его длина больше, чем *size* байт, без этого флага попытка чтения длинного сообще­ния приводит к неудаче вызова *msgrcv*. К сожалению, узнать о том, что усечение имело место, невозможно.

**Пример передачи сообщений: очередь с приоритетами**

В этом разделе разработаем простое приложение для передачи сообщений. Его целью является реализация очереди, в которой для каждого элемента может быть задан приоритет. Серверный процесс будет выбирать элементы из очереди и об­рабатывать их каким-либо образом. Например, элементы очереди могут быть име­нами файлов, а серверный процесс может копировать их на принтер. Этот пример аналогичен примеру использования FIFO.

Отправной точкой будет следующий заголовочный файл *q.h*:

*/\* q.h – заголовок для примера очереди сообщений \*/*

*#include <sys/types.h>*

*#include <sys/ipc.h>*

*#include <sys/msg.h>*

*#include <string.h>*

*#include <errno.h>*

*#define QKEY (key\_t) 0105 /\* Ключочереди \*/*

*#defineQPERM 0660 /\* Права доступа \*/*

*#defineMAXOBN 50 /\* Макс. длина имени объекта \*/*

*#define MAXPRIOR 10 /\* Максимальныйприоритет \*/*

*struct q\_entry*

*{*

*long mtype;*

*charmtext [MAXOBN+1];*

*};*

Первая часть этого файла содержит необходимые включаемые файлы. Определение QKEY задает значение ключа, которое будет обозначать очередь сообщений в системе. Определение QPERM устанавливает связанные с очередью права доступа. Так как код доступа равен 0660, то владелец очереди и члены его группы смогут выполнять чтение и запись. Как увидим позже, определения MAXOBN и MAXPRIOR будут налагать ограничения на сообщения, помещаемые в очередь. Последняя часть этого включаемого файла содержит определение структуры q\_entry. Структуры этого типа будут использоваться в качестве сообщений, передаваемых и принимаемых следующими процедурами.

Первая рассматриваемая процедура называется *enter*, она помещает в очередь имя объекта, заканчивающееся нулевым символом, и имеет следующую форму:

*/\* Процедура enter – поместить объект в очередь \*/*

*#include “q.h”*

*int enter (char \*objname, int priority)*

*{*

*intlen, s\_qid;*

*structq\_entrys\_entry; /\* Структура для хранения сообщений \*/*

*/\* Проверка длины имени и уровня приоритета \*/*

*if( (len = strlen(objname)) > MAXOBN )*

*{*

*warn (“ Слишком длинное имя “);*

*return (-1);*

*}*

*if (priority>MAXPRIOR || priority<0 )*

*{*

*warn (“ Недопустимый уровень приоритета ”);*

*return (-1);*

*}*

*/\* Инициализация очереди сообщений, если это необходимо \*/*

*if( (s\_qid = init\_queue() ) == -1)*

*return (-1);*

*/\* Инициализация структуры s\_entry \*/*

*s\_entry.mtype = (long)priority;*

*strcpy ( s\_entry.mtext, objname, MAXOBN );*

*/\* Посылаем сообщение, выполнив ожидание, если это необходимо\*/*

*if (msgsnd(s\_qid, &s\_entry, len, 0) == -1)*

*{*

*perror(“Ошибкавызова msgsnd”);*

*return (-1);*

*}*

*else*

*return (0);*

*}*

Первое действие, выполняемое процедурой *enter*, заключается в проверке длины имени объекта и уровня приоритета. Обратите внимание на то, что мини­мальное значение переменной приоритета *priority* равно 1, так как нулевое зна­чение приведет к неудачному завершению вызова *msgsnd*. Затем процедура *enter* «открывает» очередь, вызывая процедуру *init\_gueue*, реализацию которой при­ведем позже.

После завершения этих действий процедура формирует сообщение и пытается послать его при помощи вызова *msgsnd*. Здесь для хранения сообщения исполь­зована структура *s\_entry* типа *q\_entry*, и последний параметр вызова *msgsnd* равен нулю. Это означает, что система приостановит выполнение текущего про­цесса, если очередь заполнена (так как не задан флаг *IPC\_NOWAIT***).**

Процедура *enter* сообщает о возникших проблемах при помощи функции *warn* или библиотечной функции *perror*. Для простоты функция *warn* реализо­вана следующим образом:

*#include<stdio.h>*

*int warn(char \*s)*

*{*

*fprintf ( stderr, "Предупреждение: %s\n", s);*

*}*

В реальных системах функция *warn* должна записывать сообщения в специ­альный файл протокола.

Назначение функции *init\_queue* очевидно. Она инициализирует идентифи­катор очереди сообщений или возвращает идентификатор очереди сообщений, ко­торый с ней уже связан.

*/\* Инициализация очереди — получить идентификатор очереди \*/*

*#include "q.h"*

*int init\_queue(void)*

*{*

*intqueue\_id;*

*/\* Попытка создания или открытия очереди сообщений \*/*

*if ( (queue\_id = msgget(QKEY, IPC\_CREAT | QPERM)) == -1)*

*perror(“ Ошибкавызова msgget”);*

*return (queue\_id);*

*}*

Следующая процедура, *serve*, используется серверным процессом для получения сообщений из очереди и противоположна процедуре *enter*.

*/\* Процедура serve — принимает и обрабатывает сообщение очереди с наивысшим приоритетом \*/*

*#include "q.h"*

*int serve(void)*

*{*

*int mlen, r\_qid;*

*struct q\_entry r\_entry;*

*/\* Инициализация очереди сообщений, если это необходимо \*/*

*if((r\_qid = init\_queue()) == -1)*

*return (-1);*

*/\* Получить и обработать следующее сообщение \*/*

*for(;;)*

*{*

*if((mlen = msgrcv(r\_qid, &r\_entry, MAXOBN,*

*(-1 \* MAXPRIOR), MSG\_NOERROR)) == -1)*

*{*

*perror("Ошибкавызова msgrcv");*

*return (-1) ;*

*}*

*else*

*{*

*/\* Убедиться, что это строка \*/*

*r\_entry.mtext[mlen]='\0';*

*/\* Обработать имя объекта \*/*

*proc\_obj(&r\_entry);*

*}*

*}*

*}*

Обратите внимание на вызов *msgrcv*. Так как в качестве параметра типа зада­но отрицательное значение (-1 \* *MAXPRIOR*), то система вначале проверяет оче­редь на наличие сообщений со значением *mtype* равным 1, затем равным 2 и так далее, до значения *MAXPRIOR*включительно. Другими словами, сообщения с наи­меньшим номером будут иметь наивысший приоритет. Процедура *proc\_obj*работает с объектом. Для системы печати она может просто копировать файл на принтер.

Две следующих простых программы демонстрируют взаимодействие этих процедур: программа *etest* помещает элемент в очередь, а программа *stest* обрабатывает его (в действительности она всего лишь выводит содержимое и тип сообщения).

**Программа *etest***

*/\* Программа etest - ввод имен объектов в очередь \*/*

*#include <stdio.h>*

*#include <stdlib.h>*

*#include "q.h"*

*main(int argc, char \*\*argv)*

*{*

*int priority;*

*if(argc != 3 )*

*{*

*fprintf(stderr, "Применение: %s имяприоритет\n, argv[0]);*

*exit(1);*

*}*

*if((priority = atoi(argv[2])) <= 0 || priority > MAXPRIOR)*

*{*

*warn("Недопустимыйприоритет");*

*exit(2);*

*}*

*if(enter(argv[1], priority) < 0)*

*{*

*warn("Ошибка в процедуре enter");*

*exit(3);*

*}*

*exit(0) ;*

*}*

**Программа *stest***

*/\* Программа stest - простой сервер для очереди \*/*

*#include <stdio.h>*

*#include "q.h"*

*main()*

*{*

*pid\_t pid;*

*switch(pid = fork())*

*{*

*case 0: /\* Дочерний процесс \*/*

*serve();*

*break; /\* Сервер не существует \*/*

*case -1:*

*warn(“Не удалось запустить сервер”);*

*break;*

*default:*

*printf(“Серверный процесс с идентификатором %d\n”, pid);*

*}*

*exit( pid != -1 ? 0 : 1);*

*}*

*int proc\_obj( struct q\_entry \*msg)*

*{*

*printf(“\nПриоритет: %ld имя: %s\n”, msg->mtype, msg->mtext);*

*}*

Ниже следует пример использования этих двух простых программ. Перед за­пуском программы *stest*в очередь вводятся четыре простых сообщения при по­мощи программы *etest*. Обратите внимание на порядок, в котором выводятся сообщения:

$ etestobjname1 3

$ etest objname2 4

$ etest objname3 1

$ etest objname4 9

$ etest

Серверный процесс с идентификатором 2545

$

Приоритет 1 имя objname3

Приоритет 3 имя objname1

Приоритет 4 имя objname2

Приоритет 9 имя objname4

**Системный вызов msgctl**

Процедура *msgctl*служит трем целям: она позволяет процессу получать ин­формацию о статусе очереди сообщений, изменять некоторые из связанных с оче­редью ограничений или удалять очередь из системы.

***Описание***

*#include <sys/msg.h>*

*int msgctl(int mqid, int command, struct msqid\_ds \*msq\_stat);*

Переменная *mqid*должна быть допустимым идентификатором очереди. Про­пуская пока параметр *command*, обратимся к третьему параметру *msq\_stat*, кото­рый содержит адрес структуры *msqid\_ds*. Эта структура определяется в файле *<sys/msg.h>*и содержит следующие элементы:

|  |  |  |  |
| --- | --- | --- | --- |
| *struct ipc\_perm msg\_perm;* | | */\** | *Владелец/права доступа \*/* |
| *msgqnum\_t* | *msg\_qnum;* | */\** | *Число сообщений в очереди \*/* |
| *msglen\_t* | *msg\_qbytes;* | */\** | *Макс. число байтов в очереди \*/* |
| *pid\_t* | *msg\_lspid;* | */\** | *Идентификатор процесса,* |
|  |  | *последним вызвавшего msgsnd \*/* | |
| *pid\_t* | *msg\_lrpid;* | */\** | *Идентификатор процесса,* |
|  |  | *последним вызвавшего msgrcv \*/* | |
| *time\_t* | *msg\_stime;* | */\** | *Время посл. вызова msgsnd \*/* |
| *time\_t* | *msg\_rtime;* | */\** | *Время посл. вызова msgrcv \*/* |
| *time\_t* | *msg\_ctime;* | */\** | *Время посл. изменения \*/* |

Структуры *ipc\_perm*, с которыми уже встречались ранее, содержат связанную с очередью информацию о владельце и правах доступа. Типы *msgqnum*\_*t*, *msglen\_t, pid\_t*и *time\_t*зависят от конкретной системы. Переменные типа *time\_t*содер­жат число секунд, прошедшее с 00:00 по гринвичскому времени 1 января 1970 г. (Следующий пример покажет, как можно преобразовать такие значения в удобо­читаемый формат.)

Параметр *command*в вызове *msgctl*сообщает системе, какую операцию она должна выполнить. Существуют три возможных значения этого параметра, каж­дое из которых может быть применено к одному из трех средств межпроцессного взаимодействия. Они обозначаются следующими константами, определенными в файле *<sys/ipc.h>.*

*IPC\_STAT*Сообщает системе, что нужно поместить информацию о стату­се объекта в структуру *msg\_stat.*

*IPC\_SET*Используется для задания значений управляющих параметров очереди сообщений, содержащихся в структуре msg\_stat. При этом могут быть изменены только следующие поля:

*msq\_stat.msg\_perm.uid*

*msq\_stat.msg\_perm.gid*

*msq\_stat.msg\_perm.mode*

*msq\_stat.msg\_qbytes*

Операция IPC\_SETзавершится успехом только в случае ее вы­полнения суперпользователем или текущим владельцем очере­ди, заданным параметром msq\_stat .msg\_perm.uid. Кроме того, только суперпользователь может увеличивать значение msg\_qbytes- максимальное количество байтов, которое мо­жет находиться в очереди

*IPC\_RMID*Эта операция удаляет очередь сообщений из системы. Она так­же может быть выполнена только суперпользователем или вла­дельцем очереди. Если параметр *command*принимает значение *IPC\_RMID*, то параметр *msg\_stat*задается равным *NULL*

Следующий пример, программа *show\_msg*, выводит часть информации о ста­тусе объекта очереди сообщений. Программа должна вызываться так:

*$ show\_msg значение\_ключа*

Программа *show\_msg*использует библиотечную процедуру *ctime*для преоб­разования значений структуры *time\_t*в привычную запись.

Текст программы *show\_msg*:

*/\* Программа showmsg - выводит данные об очереди сообщений \*/*

*#include <sys/types.h>*

*#include <sys/ipc.h>*

*#include <sys/msg.h>*

*#include <stdio.h>*

*#include <time.h>*

*void mqstat\_print (key\_t, int, struct msqid\_ds \*);*

*main (int argc, char \*\*argv)*

*{*

*key\_t mkey;*

*int msq\_id;*

*struct msqid\_ds msq\_status;*

*if(argc != 2)*

*{*

*fprintf(stderr, "Применение: showmsgзначение\_ключа\п");*

*exit(1) ;*

*}*

*/\* Получаем идентификатор очереди сообщений \*/*

*mkey = (key\_t)atoi(argv[1]);*

*if(( msq\_id = msgget(mkey, 0)) == -1)*

*{*

*perror("Ошибкавызова msgget");*

*exit(2);*

*}*

*/\* Получаеминформациюостатусе \*/*

*if(msgctl(msq\_id, IPC\_STAT, &msq\_status) == -1)*

*{*

*perror("Ошибкавызова msgctl");*

*exit(3);*

*}*

*/\* Выводиминформациюостатусе \*/*

*mqstat\_print(mkey, msq\_id, &msq\_status)*

*exit(0);*

*}*

*void mqstat\_print(key\_t mkey, int mqid, struct msqld\_ds \*mstat)*

*{*

*printf ("\пКлюч %d, msg\_qid %d\n\n", mkey, mqid);*

*printf ("%d сообщений в очереди\п\п", mstat->msg\_qnum);*

*printf ("Последнее сообщение послано процессом %d в %s\n",*

*mstat->msg\_lspid, ctime(&(mstat->msg\_stime)));*

*printf ("Последнее сообщение принято процессом %d в %s\n",*

*mstat->msg\_lrpid, ctime(&(mstat->msg\_rtime)));*

*}*

.3. Семафоры

**Семафоры как теоретическая конструкция**

В информатике понятие *семафор*(semaphore) было впервые введено голланд­ским теоретиком Е.В. Дейкстрой (E.W. Dijkstra) для решения задач синхрониза­ции процессов. Семафор *sem*может рассматриваться как целочисленная перемен­ная, для которой определены следующие операции:

*p(sem) или wait (sem)*

*if (sem !=0)*

*уменьшить sem на единицу*

*else*

*ждать, пока sem не станет ненулевым, затем вычесть единицу*

*v(sem) или signal (sem)*

*увеличить sem на единицу*

*if (очередь ожидающих процессов не пуста)*

*продолжить выполнение первого процесса в очереди ожидания*

Обратите внимание, что обозначения *р* и *v*происходят от голландских терми­нов для понятий *ожидания*(wait) и *сигнализации*(signal), причем последнее по­нятие не следует путать с обычными сигналами UNIX.

Действия проверки и установки в обеих операциях должны составлять одно атомарное действие, чтобы только один процесс мог изменять семафор *sem*в каж­дый момент времени.

**Системный вызов semget**

**Описание**

*#include<sys\sem.h>*

*int semget(key\_t key, int nsems, int permflags);*

Вызов *semget* аналогичен вызову *msgget*. Дополнительный параметр *nsems* задает требуемое число семафоров в наборе семафоров; это важный момент - семафорные операции в *System V IPC* приспособлены для работы с наборами семафоров, а не с отдельными объектами семафоров. На рис. 8.2 показан набор семафоров. Ниже увидим, что использование целого набора семафоров усложняет интерфейс процедур работы с семафорами.

Значение, возвращаемое в результате успешного вызова *semget*, является *идентификатором набора семафоров*(semaphoresetidentifier), который ведет себя

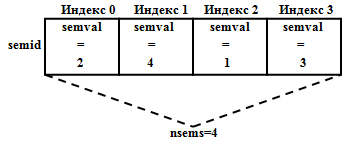


Рис. 4.1. Набор семафоров

почти так же, как идентификатор очереди сообщений. Идентификатор набора семафоров обозначен на рис. 4.1 как *semid*. Следуя обычной практике языка С, ин­декс семафора в наборе может принимать значения от 0 до *nsems-1*.

С каждым семафором в наборе связаны следующие значения:

semvalЗначение семафора, положительное целое число. Устанавливается при помощи системных вызовов работы с семафорами, то есть к зна­чениям семафоров нельзя получить прямой доступ из программы, как к другим объектам данных.

sempidИдентификатор процесса, который последним работал с семафором.

semcntЧисло процессов, ожидающих увеличения значения семафора.

semzcntЧисло процессов, ожидающих обнуления значения семафора.

**Системный вызов semctl**

**Описание**

*#include<sys/sem.h>*

*int semctl (int semid, int sem\_num, int command, union semun ctl\_arg);*

Из определения видно, что функция *semctl* намного сложнее, чем *msgctl*. Па­раметр *semid* должен быть допустимым идентификатором семафора, возвращен­ным вызовом *semget*. Параметр *command*имеет тот же смысл, что и в вызове *msgctl*, - задает требуемую команду. Команды распадаются на три категории: стандартные команды управления средством межпроцессного взаимодействия (такие как *IPC\_STAT*); команды, которые воздействуют только на один семафор; и команды, действующие на весь набор семафоров. Все доступные команды при­ведены в табл. 8.1.

Таблица 8.1. Коды функций вызова semctl

|  |
| --- |
| **Стандартные функции межпроцессного взаимодействия** |
| IPC\_STAT Поместить информацию о статусе в поле ctl\_arg.buf  IPC\_SET Установить данные о владельце/правах доступа  IPC\_RMID Удалить набор семафоров из системы |
| **Операции над одиночными семафорами**  (относятся к семафору sem\_num, значение возвращается вызовом semctl) |
| GETVAL Вернуть значение семафора ( то есть semval )  SETVAL Установить значение семафора равным ctl\_arg.val  GETPID Вернуть значение sempid  GETNCNT Вернуть semncnt (см. выше)  GETZCNT Вернуть semzcnt (см. выше) |
| **Операции над всеми семафорами** |
| GETALL Поместить значение semval в массив ctl\_arg.array  SETALL Установить все значения semval из массива ctl\_arg.array |

Параметр *sem\_num* используется со второй группой возможных операций вы­зова *semctl* для задания определенного семафора. Последний параметр *ctl\_arg* является объединением, определенным следующим образом:

unionsemun

{

int val;

struct semid\_ds \*buf;

unsignedshort \*array;

};

Каждый элемент объединения представляет некоторый тип значения, переда­ваемого вызову *semctl* при выполнении определенной команды. Например, если значение command равно *SETVAL*, то будет использоваться элемент *ctl\_arg.val*.

Одно из важных применений функции *setval* заключается в установке на­чальных значений семафоров, так как вызов *semget* не позволяет процессу сделать это. Приведенная в качестве примера функция *initsem* может использовать­ся для создания одиночного семафора и получения связанного с ним иденти­фикатора набора семафоров. После создания семафора (если семафор еще не существовал) функция *semctl* присваивает ему начальное значение равное единице.

*/\* Функция initsem - инициализация семафора \*/*

*#include "pv.h"*

*int initsem(key\_t semkey)*

*{*

*int status = 0, semid;*

*if ( (semid = semget(semkey, 1, SEMPERM|IPC\_CREAT|IPC\_EXCL)) == -1)*

*{*

*if(errno = EEXIST)*

*semid = semget(semkey, 1, 0) ;*

*}*

*else /\* Еслисемафорсоздается \*/*

*{*

*union semun arg;*

*arg.val = 1;*

*status = semctl(semid, 0, SETVAL, arg);*

*}*

*if(semid == -1|| status == -1)*

*{*

*perror("Ошибкавызова initsem");*

*return (-1);*

*}*

*/\* Все в порядке \*/*

*return (semid);*

*}*

Включаемый файл *pv. h* содержит следующие определения:

*/\* Заголовочный файл для примера работы с семафорами \*/*

*#include <sys/types.h>*

*#include <sys/ipc.h>*

*#include <sys/sem.h>*

*#include <errno.h>*

*#define SEMPERM 0600*

*#define TRUE 1*

*#define FALSE 0*

*typedef union \_semun*

*{*

*int val;*

*struct semid\_ds \*buf;*

*ushort \*array;*

*} semun;*

**Операции над семафорами: вызов semop**

Вызов *semop* выполняет основные операции над семафорами.

**Описание**

*#include <sys/sem.h>*

*int semop(int semid, struct sembuf \*op\_array, size\_t num\_ops);*

Переменная *semid* является идентификатором набора семафоров, полученным с помощью вызова *semget*. Параметр *op\_array*является массивом струк­тур *sembuf*, определенных в файле *<sys/sem.h>.* Каждая структура *sembuf* со­держит описание операций, выполняемых над семафором.

И снова основной акцент делается на операции с наборами семафоров, при этом функция *semop* позволяет выполнять группу операций как атомарную опе­рацию. Это означает, что пока не появится возможность одновременного выпол­нения всех операций с отдельными семафорами набора, не будет выполнена ни одна из этих операций. Если не указано обратного, процесс приостановит работу дотех пор, пока он не сможет выполнить все операции сразу.

Рассмотрим структуру *sembuf*. Она включает в себя следующие элементы: *unsignedshortsem\_num;*

*short sem\_op;*

*short sem\_flg;*

Поле *sem\_num* содержит индекс семафора в наборе. Если, например, набор со­держит всего один элемент, то значение *sem\_num* должно быть равно нулю. Поле *sem\_op* содержит целое число со знаком, значение которого сообщает функции *semop*, что необходимо сделать. При этом возможны три случая:

**Случай 1: отрицательное значение *sem\_op***

Это обобщенная форма команды для работы с семафорами p (), которая обсуждалась ранее. Действие функции *semop* можно описать при помощи псевдокода следующим образом (обратите внимание, что *ABS()* обозначает модуль пере­менной):

*if( semval >= ABS(sem\_op) )*

*{*

*semval = semval - ABS(sem\_op);*

*}*

*else*

*{*

*if( (sem\_flg & IPC\_NOWAIT) )*

*немедленно вернуть -1*

*else*

*{*

*ждать, пока semval не станет больше или равно ABS(sem\_op)*

*затем, как и выше, вычесть ABS(sem\_op)*

*}*

*}*

Основная идея заключается в том, что функция *semop* вначале проверяет значе­ние *semval*, связанное с семафором *sem\_num*. Если значение *semval* достаточно ве­лико, то оно сразу уменьшается на указанную величину. В противном случае процесс будет ждать, пока значение *semval* не станет достаточно большим. Тем не менее, если в переменной *sem\_flg* установлен флаг *IPC\_NOWAIT*, то возврат из вызова *sem\_op* произойдет немедленно, и переменная *errno* будет содержать код ошибки *EAGAIN*.

**Случай 2: положительное значение sem\_op**

Это соответствует традиционной операции v (). Значение переменной *sem\_op* просто прибавляется к соответствующему значению *semval*. Если есть процессы, ожидающие изменения значения этого семафора, то они могут продолжить вы­полнение, если новое значение семафора удовлетворит их условия.

**Случай 3: нулевое значение *sem\_op***

В этом случае вызов *sem\_op* будет ждать, пока значение семафора не станет равным нулю; значение *semval* этим вызовом не будет изменяться. Если в пере­менной *sem\_flg* установлен флаг *IPC\_NOWAIT*, а значение *semval* еще не равно нулю, то функция *semop* сразу же вернет сообщение об ошибке.

**Флаг *SEM\_UNDO***

Это еще один флаг, который может быть установлен в элементе *sem\_flg* струк­туры *sembuf*. Он сообщает системе, что нужно автоматически «отменить» эту опе­рацию после завершения процесса. Для отслеживания всей последовательности таких операций система поддерживает для семафора целочисленную переменную *semadj*. Важно понимать, что переменная *semadj* связана с процессами, и для раз­ных процессов один и тот же семафор будет иметь различные значения *semadj*. Если при выполнении операции *semop* установлен флаг *SEM\_UNDO*, то значение переменной *sem\_num* просто вычитается из значения *semadj*. При этом важен знак переменной *sem\_num*: значение *semadj* уменьшается, если значение *sem\_num*положительное, и увеличивается, если оно отрицательное. После выхода из про­цесса система прибавляет все значения *semadj* к соответствующим семафорам и, таким образом, сводит на нет эффект от всех вызовов *semop*. В общем случае флаг *SEM\_UNDO*должен быть всегда установлен, кроме тех случаев, когда значения, уста­навливаемые процессом, должны сохраняться после завершения процесса.

**Пример работы с семафорами**

Теперь продолжим пример, который начали с процедуры *initsem*. Он содержит две процедуры p() и v(), реализующие традиционные операции над семафо­рами. Сначала рассмотрим p():

*/\* Процедура р.с - операция р для семафора \*/*

*#include "pv.h"*

*int р (int semid)*

*{*

*struct sembuf p\_buf;*

*p\_buf.sem\_num = 0;*

*p\_buf.sem\_op = -1;*

*p\_buf.sem\_fig = SEM\_UNDO;*

*if(semop(semid, &p\_buf, 1) == -1)*

*{*

*perror("Ошибка операции p(semid)");*

*exit(1);*

*}*

*return (0);*

*}*

Обратите внимание на то, что здесь использован флаг *SEM\_UNDO*. Теперь рас­смотрим текст процедуры v().

*/\* Процедура v.с - операция v для семафора \*/*

*#include "pv.h"*

*int v(int semid)*

*{*

*struct sembuf v\_buf;*

*v\_buf .sem\_num = 0;*

*v\_buf.sem\_op = 1;*

*v\_buf .sem\_fig = SEM\_UNDO;*

*if (semop (semid, &v\_buf, 1) == 1)*

*{*

*perror("Ошибка операции v(semid)");*

*exit(1);*

*}*

*return (0);*

*}*

.4. Разделяемая память

Операции с разделяемой памятью позволяют двум и более процессам совмест­но использовать область физической памяти (общеизвестно, что обычно облас­ти данных любых двух программ совершенно отделены друг от друга). Чаще всего разделяемая память является наиболее производительным механизмом межпроцессного взаимодействия.

Для того, чтобы сегмент памяти мог использоваться совместно, он должен быть сначала создан при помощи системного вызова *shmget***.** После создания сегмента разделяемой памяти процесс может подключаться к нему при помощи вызова *shmat*и затем использовать его для своих частных целей. Когда этот сегмент па­мяти больше не нужен, процесс может отключиться от него при помощи вызова *shmdt*.

**Системный вызов shmget**

Сегменты разделяемой памяти создаются при помощи вызова *shmget.*

**Описание**

*#include <sys/shm.h>*

*int shmget(key\_t key, size\_t size, int permflags);*

Этот вызов аналогичен вызовам *msgget*и *semget***.** Наиболее интересным па­раметром вызова является *size*,который задает требуемый минимальный размер (в байтах) сегмента памяти. Параметр *key*является значением ключа сегмента па­мяти, параметр *permflags*задает права доступа к сегменту памяти и, кроме того, может содержать флаги *IPC\_CREAT*и *IPC\_EXCL*.

**Операции с разделяемой памятью: вызовы shmat и shmdt**

Сегмент памяти, созданный вызовом *shmget,*является участком *физической* памяти и не находится в *логическом* пространстве данных процесса. Для исполь­зования разделяемой памяти текущий процесс, а также все другие процессы, вза­имодействующие с этим сегментом, должны явно подключать этот участок памя­ти к логическому адресному пространству при помощи вызова *shmat:*

**Описание**

*#include <sys/shm.h>*

*void \*shmat(int shmid, const void \*daddr, int shmflags);*

Вызов *shmat* связывает участок памяти, обозначенный идентификатором *shmid*(который был получен в результате вызова *shmget*) с некоторым допустимым адресом логического адресного пространства вызывающего процесса. Этот адрес является значением, возвращаемым вызовом *shmat* (в языке C такие адреса дан­ных обычно представляются типом void \*).

Параметр *daddr* позволяет программисту до некоторой степени управлять вы­бором этого адреса. Если этот параметр равен *NULL*, то участок подключается к первому доступному адресу, выбранному системой. Это наиболее простой слу­чай использования вызова *shmat*. Если параметр *daddr* не равен *NULL*, то участок будет подключен к содержащемуся в нем адресу или адресу в ближайшей окрест­ности в зависимости от флагов, заданных в аргументе *shmflags*. Этот вариант сложнее, так как при этом необходимо знать расположение программы в памяти.

Аргумент *shmflag* может содержать два флага, *SHM\_RDONLY*и *SHM\_RND*, опре­деленные в заголовочном файле *<sys/shm.h>.* При задании флага *SHM\_RDONLY*участок памяти подключается только для чтения. Флаг *SHM\_RMD*определяет, если это возможно, способ обработки в вызове *shmat* ненулевого значения *daddr*.

В случае ошибки вызов *shmat* вернет значение:

*(void\*)-1*

Вызов *shmdt* противоположен вызову *shmat* и отключает участок разделяемой памяти от логического адресного пространства процесса (это означает, что про­цесс больше не может использовать его). Он вызывается очень просто:

*retval = shmdt(memptr);*

Возвращаемое значение *retval* является целым числом и равно 0 в случае ус­пеха и -1 - в случае ошибки.

**Системный вызов *shmctl***

**Описание**

*#include<sys/shm.h>*

*int shmctl(int shmid, int command, struct shmid ds \*shm\_stat);*

Этот вызов в точности соответствует вызову *msgctl*, и параметр *command* мо­жет, наряду с другими, принимать значения *IPC\_STAT*, *IPC\_SET*и *IPC\_RMID*. В следующем примере этот вызов будет использован с аргументом *command* рав­ным *IPC\_RMID*.

**Пример работы с разделяемой памятью: программа *shmcopy***

В этом разделе создадим простую программу *shmcopy* для демонстрации прак­тического использования разделяемой памяти. Программа *shmcopy* просто копи­рует данные со своего стандартного ввода на стандартный вывод, но позволяет избежать лишних простоев в вызовах *read* и *write*. При запуске программы *shmcopy* создаются два процесса, один из которых выполняет чтение, а другой - запись, и которые совместно используют два буфера, реализованные в виде сег­ментов разделяемой памяти. Когда первый процесс считывает данные в первый буфер, второй записывает содержимое второго буфера, и наоборот. Так как чтение и запись выполняются одновременно, пропускная способность возрастает. Этот подход используется, например, в программах, которые выводят информа­цию на ленточный накопитель.

Для согласования двух процессов (чтобы записывающий процесс не писал в буфер до тех пор, пока считывающий процесс его не заполнит) будем использо­вать два семафора. Почти во всех программах, использующих разделяемую па­мять, требуется дополнительная синхронизация, так как механизм разделяемой памяти не содержит собственных средств синхронизации.

Программа *shmcopy* использует следующий заголовочный файл *share\_ex.h:/\* Заголовочный файл для примера работы с разделяемой памятью \*/*

*#include <stdio.h>*

*#include <signal.h>*

*#include <sys/types.h>*

*#include <sys/ipc.h>*

*#include <sys/shm.h>*

*#include <sys/sem.h>*

*#define SHMKEY1 (key\_t)0x10 /\* Ключразделяемойпамяти \*/*

*#define SHMKEY2 (key\_t)0x15 /\* Ключразделяемойпамяти \*/*

*#define SEMKEY (key\_t)0x20 /\* Ключсемафора \*/*

*/\* Размер буфера для чтения и записи \*/*

*#defineSIZ 5\*BUFSIZ*

*/\* В этой структуре будут находиться данные и счетчик чтения \*/*

*struct databuf*

*{*

*int d\_nread;*

*char d\_buf[SIZ];*

*};*

*typedef union semun*

*{*

*int val;*

*struct semid\_ds \*buf;*

*ushort \*array;*

*} semun;*

Напомним, что постоянная *BUFSIZ*определена в файле *<stdio.h>*и задает оптимальный размер порций данных при работе с файловой системой. Шаблон *databuf*показывает структуру, которая связывается с каждым сегментом раз­деляемой памяти. В частности, элемент *d\_nread* позволит процессу, выполняю­щему чтение, передавать другому, осуществляющему запись, через участок разде­ляемой памяти число считанных символов.

Следующий файл содержит процедуры для инициализации двух участков раз­деляемой памяти и набора семафоров. Он также содержит процедуру *remobj*, ко­торая удаляет различные объекты межпроцессного взаимодействия в конце рабо­ты программы. Обратите внимание на способ вызова *shmat*для подключения участков разделяемой памяти к адресному пространству процесса.

*/\* Процедуры инициализации \*/*

*#include "share\_ex.h"*

*#define IFLAGS (IPC\_CREAT | IPC\_EXCL)*

*#define ERR ((struct databuf \*) -1)*

*static int shmid1, shmid2, semid;*

*void getseg(struct databuf \*\*p1, struct databuf \*\*p2)*

*{*

*/\* Создатьучастокразделяемойпамяти \*/*

*if((shmid1 = shmget(SHMKEY1, sizeof(struct databuf),*

*0600 | IFLAGS)) == -1)*

*fatal("shmget");*

*if((shmid2 = shmget(SHMKEY2, sizeof(struct databuf),*

*0600 | IFLAGS)) == -1)*

*fatal("shmget");*

*/\* Подключить участки разделяемой памяти \*/*

*if((\*p1 = (struct databuf \*)shmat(shmid1,0,0)) == ERR)*

*fatal ("shmat");*

*if((\*p2 = (struct databuf \*)shmat(shmid2,0,0)) == ERR)*

*fatal ("shmat");*

*}*

*int getsem(void) /\* Получитьнаборсемафоров \*/*

*{*

*union semun x;*

*x.val = 0;*

*/\* Создатьдванаборасемафоров \*/*

*if((semid = semget(SEMKEY, 2, 0600 | IFLAGS)) == -1)*

*fatal("semget");*

*/\* Задать начальные значения \*/*

*if(semctl(semid, 0, SETVAL, x) == -1)*

*fatal("semctl");*

*if(semctl(semid, 1, SETVAL, x) == -1)*

*fatal("semctl");*

*return(semid);*

*}*

*/\* Удалить идентификаторы разделяемой памяти*

*и идентификатор набора семафоров \*/*

*voidremobj(void)*

*{*

*if(shmctl(shmid1, IPC\_RMID, NULL) == -1)*

*fatal("shmctl");*

*if(shmctl(shmid2, IPC\_RMID, NULL) == -1)*

*fatal("shmctl");*

*if(semctl(semid, 0, IPC\_RMID, NULL) == -1)*

*fatal("semctl");*

*}*

Ошибки в этих процедурах обрабатываются при помощи процедуры *fatal*, ко­торая использовалась в предыдущих примерах. Она просто вызывает *perror*, а затем *exit*.

Ниже следует функция *main* для программы *strcopy*. Она вызывает процедуры инициализации, а затем создает процесс для чтения (родительский) и для за­писи (дочерний). Обратите внимание на то, что именно выполняющий запись процесс вызывает процедуру remobj при завершении программы.

*/\* Программа shmcopy - функция main \*/*

*#nclude "share\_ex.h"*

*main()*

*{*

*int semid;*

*pid\_t pid;*

*structdatabuf \*buf1, \*buf2;*

*/\* Инициализация набора семафоров \*/*

*semid = getsem();*

*/\* Создать и подключить участки разделяемой памяти \*/*

*getseg(&buf1, &buf2);*

*switch (pid = fork()){*

*case -1:*

*fatal("fork");*

*case 0: /\* Дочернийпроцесс \*/*

*writer(semid, buf1, buf2);*

*remobj() ;*

*break;*

*default: /\* Родительскийпроцесс \*/*

*reader(semid, buf1, buf2);*

*break;*

*}*

*exit (0);*

*}*

Функция *main* создает объекты межпроцессного взаимодействия до вызова *fork*. Обратите внимание на то, что адреса, определяющие сегменты разделяемой памяти (которые находятся в переменных buf1 и buf2), будут заданы в обоих про­цессах.

Процедура *reader* принимает данные со стандартного ввода, то есть из дескриптора файла 0, и является первой функцией, представляющей интерес. Ей пе­редается идентификатор набора семафоров в параметре *semid* и адреса двух уча­стков разделяемой памяти в переменных buf1 и buf2.

*/\* Процедура reader - выполняет чтение из файла \*/*

*#include "share\_ex.h"*

*/\* Определения процедур р() и v() для двух семафоров \*/*

*struct sembuf p1 = {0,-1,0}, p2 = {1,-1,0};*

*struct sembuf v1 = {0,1,0}, v2 = {1,1,0};*

*void reader(int semid, struct databuf \*buf1,*

*struct databuf \*buf2)*

*{*

*for(;;)*

*{*

*/\* Считатьвбуфер buf1 \*/*

*buf1->d\_nread = read(0, buf1->d\_buf, SIZ);*

*/\* Точка синхронизации \*/*

*semop(semid, &v1, 1);*

*semop(semid, &p2, 1);*

*/\* Чтобы процедура writer не была приостановлена \*/*

*if (buf1->d\_nread <=0)*

*return;*

*buf2->d\_nread = read(0, buf2->d\_buf, SIZ);*

*semop(semid, &v2, 1);*

*semop(semid, &p1, 1);*

*if (buf2->d\_nread <=0)*

*return;*

*}*

*}*

Структуры *sembuf* просто определяют операции p () и v () для набора из двух семафоров. Но на этот раз они используются не для блокировки критических участков кода, а для синхронизации процедур, выполняющих чтение и запись. Процедура reader использует операцию v2 для сообщения о том, что она завершила чтение и ожидает, вызвав *semop* с параметром p1, пока процедура *writer* не сообщит о завершении записи. Это станет более очевидным при описании проце­дуры *writer*. Возможны другие подходы, включающие или четыре бинарных се­мафора, или семафоры, имеющие более двух значений.

Последней процедурой, вызываемой программой *shmcopy*, является процедура *writer*:

*/\* Процедура writer - выполняет запись \*/*

*#include "share\_ex.h"*

*externstructsembufp1, p2; /\* Определены в reader.с \*/*

*extern struct sembuf v1, v2; /\* Определеныв reader.с \*/*

*void writer(int semid, struct databuf \*buf1, struct databuf \*buf2)*

*{*

*for(;;)*

*{*

*semop(semid, &p1, 1);*

*semop(semid, &v2, 1);*

*if(buf1->d\_nread <= 0)*

*return;*

*write(1, buf1->d\_buf, buf1->d\_nread) ;*

*semop(semid, &p2, 1);*

*semop(semid, &v1, 1);*

*if(buf2->d\_nread <= 0)*

*return;*

*write(1, buf2->d\_buf, buf2->d\_nread) ;*

*}*

*}*

И снова следует обратить внимание на использование набора семафоров для согласования работы процедур *reader* и *writer*. На этот раз процедура *writer* использует операцию v2 для сигнализации и ждет p1. Важно также отметить, что значения bufl->d\_nread и buf2->d\_nread устанавливаются процессом, выполняющим чтение.

После компиляции можно использовать программу *shmcopy* при помощи по­добной команды:

*$ shmcopy<big> /tmp/big.*

***Тема 5.*** ***Понятие потока в ОС UNIX***

### Различие между процессами и потокам

С помощью процессов можно организовать параллельное выполнение программ. Для этого процессы размножаются вызовами fork(), а затем между ними организуется взаимодействие с помощью средствами межпроцессного взаимодействия.

Для организации параллельного выполнения и взаимодействия процессов можно использовать механизм многопоточности. Основной единицей здесь является поток. Поток представляет собой облегченную версию процесса. Основные характеристики процесса следующие.

1. Процесс располагает определенными ресурсами. Он размещен в некотором виртуальном адресном пространстве, содержащем образ этого процесса. Кроме того, процесс управляет другими ресурсами (файлы, устройства ввода / вывода и т.д.).

2. Процесс подвержен диспетчеризации. Он определяет порядок выполнения одной или нескольких программ, при этом выполнение может перекрываться другими процессами. Каждый процесс имеет состояние выполнения и приоритет диспетчеризации.

Если рассматривать эти характеристики независимо друг от друга (как это принято в современной теории ОС), то:

1) владельцу ресурса, обычно называемому процессом или задачей, присущи:

- виртуальное адресное пространство;

- индивидуальный доступ к процессору, другим процессам, файлам, и ресурсам ввода – вывода;

2) модулю для диспетчеризации, обычно называемому потоком или облегченным процессом, присущи:

- состояние выполнения (активное, готовность и т.д.);

- сохранение контекста потока в неактивном состоянии;

- стек выполнения и некоторая статическая память для локальных переменных;

- доступ к пространству памяти и ресурсам своего процесса.

Все потоки процесса разделяют общие ресурсы. Изменения, вызванные одним потоком, становятся немедленно доступны другим.

При корректной реализации потоки имеют определенные преимущества перед процессами. Им требуется:

- меньше времени для создания нового потока, поскольку создаваемый поток использует адресное пространство текущего процесса;

- меньше времени для завершения потока;

- меньше времени для переключения между двумя потоками в пределах процесса;

- меньше коммуникационных расходов, поскольку потоки разделяют все ресурсы, и в частности адресное пространство. Данные, продуцируемые одним из потоков, немедленно становятся доступными всем другим потокам.

Так как ОС UNIX является многозадачной системой, то в ходе работы несколько процессов могут конкурировать между собой за доступ к различным ресурсам. Для справедливого распределения ресурсов (память, дисковое пространство) каждому из процессов устанавливается индивидуальный набор ограничений.

### Преимушества многопоточности

Если операционная система поддерживает концепции потоков в рамках одного процесса, она называется многопоточной. Многопоточные приложения имеют ряд преимуществ.

1. Улучшенная реакция приложения - любая программа, содержащая много не зависящих друг от друга действий, может быть перепроектирована так, чтобы каждое действие выполнялось в отдельном потоке. Например, пользователь многопоточного интерфейса не должен ждать завершения одной задачи, чтобы начать выполнение другой.

2. Более эффективное использование мультипроцессирования - как правило, приложения, реализующие параллелизм через потоки, не должны учитывать число доступных процессоров. Производительность приложения равномерно увеличивается при наличии дополнительных процессоров. Численные алгоритмы и приложения с высокой степенью параллелизма, например перемножение матриц, могут выполняться намного быстрее.

3. Улучшенная структура программы - некоторые программы более эффективно представляются в виде нескольких независимых или полуавтономных единиц, чем в виде единой монолитной программы. Многопоточные программы легче адаптировать к изменениям требований пользователя.

4. Эффективное использование ресурсов системы - программы, использующие два или более процессов, которые имеют доступ к общим данным через разделяемую память, содержат более одного потока управления. При этом каждый процесс имеет полное адресное пространство и состояние в операционной системе. Стоимость создания и поддержания большого количества служебной информации делает каждый процесс более затратным, чем поток. Кроме того, разделение работы между процессами может потребовать от программиста значительных усилий, чтобы обеспечить связь между потоками в различных процессах или синхронизировать их действия.

### Уровни потоков

Существует две основных категории потоков с точки зрения реализации:

- пользовательские потоки, которые реализуются через специальные библиотеки потоков;

- потоки уровня ядра, которые реализуются через системные вызовы.

Каждый уровень имеет свои достоинства и недостатки. Некоторые операционные системы позволяют реализовать потоки обоих уровней.

Пользовательские потоки.

При использовании этого уровня ядро не знает о существовании потоков - все управление потоками реализуется приложением с помощью специальных библиотек. Переключение потоков не требует привилегий режима ядра, а планирование полностью зависит от приложения. При этом ядро управляет деятельностью процесса. Если поток вызывает системную функцию, то будет блокирован весь процесс, но для поточной библиотеки этот поток будет находиться в активном состоянии. Здесь состояние потока не зависит от состояния процесса.

Преимущества пользовательских потоков в следующем:

- переключение потоков не требует участия ядра - нет переключения из режима задачи в режим ядра;

- планирование может определяться приложением - при этом выбирается наилучший алгоритм;

- пользовательские потоки могут применяться в любой ОС - необходимо лишь наличие совместимой библиотеки потоков.

Недостатки пользовательских потоков:

- большинство системных вызовов является блокирующими и ядро блокирует процессы - включая все потоки в пределах процесса;

- ядро может направлять на процессоры только процессы - два потока в пределах одного и того же процесса не могут выполняться одновременно на двух разных процессорах.

### Потоки уровня ядра.

На этом уровне все управление потоком выполняется ядром. Используется программный интерфейс приложения (системные вызовы) для работы с потоками уровня ядра. Ядро поддерживает информацию о контексте процесса и потоков; переключение потоков требует выполнения дисциплины планирования ядра на уровне этих потоков.

Преимущества потоков уровня ядра:

- ядро может одновременно планировать выполнение нескольких потоков одного процесса на нескольких процессорах, блокирование выполняется на уровне потока;

- процедуры ядра могут быть многопоточными.

Недостатки:

- переключение потоков в пределах одного процесса требует участия ядра.

Основной библиотекой для реализации пользовательских потоков является библиотека потоков POSIX, которая называется pthreads.

### **Создание потока**

Функция *pthread\_create()* позволяет добавить новый поток управления к текущему процессу. Прототипфункции:

*int pthread\_create(pthread\_t \*tid, const pthread\_attr\_t \*tattr, void\*(\*start\_routine)(void \*), void \*arg);*

Когда атрибуты объекта не определены, они равны NULL, и поток, создаваемый по умолчанию, имеет следующие признаки: неорганиченность, неотделенность от процесса, стек с размером по умолчанию, приоритет родителя. Существует возможность также создать объект атрибутов потока с помощью функции *pthread\_attr\_init()*, а затем использовать этот объект для создания самого потока. Пример создания потока:

*#include <pthread.h>*

*pthread\_attr\_t tattr;*

*pthread\_t tid;*

*extern void \*start\_routine(void \*arg);*

*void \*arg;*

*int ret;*

*/\* поведение по умолчанию\*/*

*ret = pthread\_create(&tid, NULL, start\_routine, arg);*

*/\* инициализация с атрибутами по умолчанию \*/*

*ret = pthread\_attr\_init(&tattr);*

*/\* определение поведения по умолчанию\*/*

*ret = pthread\_create(&tid, &tattr, start\_routine, arg);*

Функция *pthread\_create()* вызывается с атрибутом *attr*, определяющим необходимое поведение; *start\_routine* - это функция, с которой новый поток начинает свое выполнение. Когда *start\_routine* завершается, поток завершается со статусом выхода, установленным в значение, возвращенное *start\_routine.*

Если вызов *pthread\_create()* успешно завершен, идентификатор созданного потока сохраняется по адресу*tid*.

Создание потока с использованием аргумента атрибутов NULL оказывает тот же эффект, что и использование атрибута по умолчанию: оба создают одинаковый поток. При инициализации *tattr* он обретает поведение по умолчанию; *pthread\_create()* возвращает 0 при успешном завершении. Любое другое значение указывает, что произошла ошибка.

### Ожидание завершения потока

Функция *pthread\_join()* используется для ожидания завершения потока:

*int pthread\_join(thread\_t tid, void \*\*status);*

Пример использования функции:

*#include <pthread.h>*

*pthread\_t tid;*

*int ret;*

*int status;*

*/\* ожидание завершения потока "tid" со статусом status \*/*

*ret = pthread\_join(tid, &status);*

*/\* ожидание завершения потока "tid" без статуса \*/*

*ret = pthread\_join(tid, NULL);*

Функция *pthread\_join()* блокирует вызывающий поток, пока указанный поток не завершится. Указанный поток должен принадлежать текущему процессу и не должен быть отделен. Если *status* не равен NULL, он указывает на переменную, которая принимает значение статуса выхода завершенного потока при успешном завершении *pthread\_join()*. Несколько потоков не могут ждать завершения одного и того же потока. Если они пытаются выполнить это, один поток завершается успешно, а все остальные - с ошибкой ESRCH. После завершения *pthread\_join()*, любое пространство стека, связанное с потоком, может быть использовано приложением.

В следующем примере один поток верхнего уровня вызывает процедуру, которая создает новый вспомогательный поток, выполняющий сложный поиск в базе данных, требующий определенных затрат времени. Главный поток ждет результатов поиска, и в то же время может выполнять другую работу. Он ждет своего помощника с помощью функции *pthread\_join()*. Аргумент*pbe*является параметром стека для нового потока.

Исходный код для thread.c:

*void mainline (...)*

*{*

*struct phonebookentry \*pbe;*

*pthread\_attr\_t tattr;*

*pthread\_t helper;*

*int status;*

*pthread\_create(&helper, NULL, fetch, &pbe);*

*/\* выполняет собственную задачу \*/*

*pthread\_join(helper, &status);*

*/\* теперь можно использовать результат \*/*

*}*

*void fetch(struct phonebookentry \*arg)*

*{*

*struct phonebookentry \*npbe;*

*/\* ищем значение в базе данных \*/*

*npbe = search (prog\_name)*

*if (npbe != NULL)*

*\*arg = \*npbe;*

*pthread\_exit(0);*

*}*

*struct phonebookentry {*

*char name[64];*

*char phonenumber[32];*

*char flags[16];*

*}*

### Отделение потока

Функция *pthread\_detach()* применяется как альтернатива *pthread\_join()*, чтобы утилизировать область памяти для потока, который был создан с атрибутом detachstate, установленным в значение PTHREAD\_CREATE\_JOINABLE.

Прототипфункции:

*int pthread\_detach(thread\_t tid);*

Пример вызова функции:

*#include <pthread.h>*

*pthread\_t tid;*

*int ret;*

*/\* отделить поток tid \*/*

*ret = pthread\_detach(tid);*

Функция *pthread\_detach()* используется, чтобы указать библиотеке потоков, что выделенная память для потока *tid*может быть утилизирована, когда поток завершится. Если tid не закончился, *pthread\_detach()* не ускоряет его завершения и возвращает 0 при успешном завершении. Любое другое значение указывает, что произошла ошибка.

### Работа с ключами потока

Однопоточные программы на C содержат два основных класса данных: локальные и глобальные. Для многопоточных программ на C добавляется третий класс: данные потока. Они похожи на глобальные данные, за исключением того, что они являются собственными для потока.

Данные потока являются единственным способом определения и обращения к данным, которые принадлежат отдельному потоку. Каждый элемент данных потока связан с ключом, который является глобальным для всех потоков процесса. Используя ключ, поток может получить доступ к указателю (void \*), который поддерживается только для этого потока.

Функция *pthread\_keycreate()* применяется для выделения ключа, который используется при идентифицикации данных некоторого потока в составе процесса. Ключ для всех потоков общий, и все потоки вначале содержат значение ключа NULL. Отдельно для каждого ключа перед его использованием вызывается *pthread\_keycreate()*. При этом не происходит никакой синхронизации. Как только ключ будет создан, каждый поток может связать с ним свое значение. Значения являются специфичными для потока и поддерживаются для каждого из них независимо. Связь ключа с потоком удаляется, когда поток заканчивается, при этом ключ должен быть создан с функцией деструктора. Прототипфункции:

*int pthread\_key\_create(pthread\_key\_t \*key, void(\*destructor)(void \*));*

Пример использования:.

*#include <pthread.h>*

*pthread\_key\_t key;*

*int ret;*

*/\* создание ключа без деструктора \*/*

*ret = pthread\_key\_create(&key, NULL);*

*/\* создание ключа с деструктором \*/*

*ret = pthread\_key\_create(&key, destructor);*

Если *pthread\_keycreate()* завершается успешно, то выделенный ключ будет сохранен в переменной*key*. Вызывающий процесс должен гарантировать, что хранение и доступ к этому ключу синхронизированы. Чтобы освободить ранее выделенную память, может использоваться дополнительная функция удаления - деструктор. Если ключ имеет непустой указатель на функцию деструктора, и поток имеет непустое значение ключа, функция деструктора вызывается для значения, связанного с потоком, после его завершения. Порядок, в котором вызываются функции деструктора, может быть произвольным;*pthread\_keycreate()* возвращает 0 при успешном завершении, или любое другое значение при возникновении ошибки.

Функция *pthread\_keydelete()* используется, чтобы уничтожить существующий ключ данных для определенного потока. Любая выделенная память, связанная с ключом, может быть освобождена, потому что ключ был удален; попытка ссылки на эту память вызовет ошибку.

Прототип*pthread\_keydelete()*:

*int pthread\_key\_delete(pthread\_key\_t key);*

Пример использования функции:

*#include <pthread.h>*

*pthread\_key\_t key;*

*int ret;*

*/\* key был создан ранее \*/*

*ret = pthread\_key\_delete(key);*

Как только ключ удален, любая ссылка на него через *pthread\_setspecific()* или *pthread\_getspecific()* приводит к ошибке EINVAL. Программист должен сам освобождать любые выделенные потоку ресурсы перед вызовом функции удаления. Эта функция не вызывает деструктора; *pthread\_keydelete()* возвращает 0 - после успешного завершения - или любое другое значение - в случае ошибки. Функция *pthread\_setspecific()* используется, чтобы установить связку между потоком и указанным ключом данных для потока. Прототипфункции:

*int pthread\_setspecific(pthread\_key\_t key, const void \*value);*

Примервызова:

*#include <pthread.h>*

*pthread\_key\_t key;*

*void \*value;*

*int ret;*

*/\* key был создан ранее \*/*

*ret = pthread\_setspecific(key, value);*

Функция *pthread\_setspecific()* возвращает 0 - после успешного завершения - или любое другое значение - в случае ошибки; она не освобождает память для хранения ключа. Если установлена новая привязка значения ключа, предыдущая привязка должна быть освобождена; иначе может произойти утечка памяти.

Чтобы получить привязку ключа для вызывающего потока, используется функция *pthread\_getspecific().* Полученное значение сохраняется в переменной *value*.

Прототипфункции:

*int pthread\_getspecific(pthread\_key\_t key);*

Пример:

*#include <pthread.h>*

*pthread\_key\_t key;*

*void \*value;*

*/\* key был создан ранее \*/*

*value = pthread\_getspecific(key);*

*Рассмотримследующийкод:*

*body() {*

*...*

*while (write(fd, buffer, size) == -1) {*

*if (errno != EINTR) {*

*fprintf(mywindow, "%s\n", strerror(errno));*

*exit(1);*

*}*

*}*

*...*

*}*

Этот код может быть выполнен любым числом потоков, но он содержит ссылки на две глобальных переменных, *errno* и *mywindow*, которые должны быть ссылками на объекты, являющиеся частными для каждого потока.

Ссылки на *errno* должны получить код системной ошибки из процедуры, вызванной именно этим конкретным потоком, и никаким другим. Поэтому ссылки на *errno* в одном потоке относятся к иной области памяти, чем ссылки на *errno* в других потоках. Переменная *mywindow* предназначена для обращения к потоку *stdio*, связанному с окном, которое является частным объектом потока. Так же, как и *errno*, ссылки на *mywindow* в одном потоке должны обращаться к отдельной конкретной области памяти (и в конечном счете - к различным окнам). Единственное различие между этими переменными состоит в том, что библиотека потоков реализует раздельный доступ для *errno*, а программист должен сам реализовать это для *mywindow*. Следующий пример показывает, как работают ссылки на *mywindow*. Препроцессор преобразует ссылки на *mywindow* в вызовы процедур *mywindow*. Эта процедура в свою очередь вызывает *pthread\_getspecific()*, передавая ему глобальную переменную *mywindow\_key* (это, действительно, глобальная переменная) и выходной параметр *win*, который принимает идентификатор окна для этого потока.

Следующийфрагменткода:

*thread\_key\_t mywin\_key;*

*FILE \*\_mywindow(void) {*

*FILE \*win;*

*pthread\_getspecific(mywin\_key, &win);*

*return(win);*

*}*

*#define mywindow \_mywindow()*

*void routine\_uses\_win( FILE \*win) {*

*...*

*}*

*void thread\_start(...) {*

*...*

*make\_mywin();*

*...*

*routine\_uses\_win( mywindow )*

*...*

*}*

Переменная *mywin\_key* определяет класс переменных, для которых каждый поток содержит собственную частную копию; т. е. эти переменные представляют собой данные этого потока. Каждый поток вызывает *make\_mywin*, чтобы инициализировать свое окно и обращаться к своему экземпляру *mywindow* для ссылки на окно. Как только эта процедура вызвана, поток может обращаться к mywindow и получать ссылку на свое частное окно. При этом ссылки на mywindow используются так, как будто они являются прямыми ссылками на частные данные потока.

Теперь можно устанавливать собственные данные потока:

*void make\_mywindow(void) {*

*FILE \*\*win;*

*static pthread\_once\_t mykeycreated =*

*PTHREAD\_ONCE\_INIT;*

*pthread\_once(&mykeycreated, mykeycreate);*

*win = malloc(sizeof(\*win));*

*create\_window(win, ...);*

*pthread\_setspecific(mywindow\_key, win);*

*}*

*void mykeycreate(void) {*

*pthread\_keycreate(&mywindow\_key, free\_key);*

*}*

*void free\_key(void \*win) {*

*free(win);*

*}*

Сначала нужно получить уникальное значение для ключа *mywin\_key*. Этот ключ используется, чтобы идентифицировать класс данных потока. Первый поток, который вызывает *make\_mywin*, вызывает также *pthread\_keycreate()*, который присваивает своему первому аргументу уникальный ключ. Функция деструктора является вторым аргументом для освобождения экземпляра определенного элемента данных в потоке, как только этот поток завершится.

Следующий шаг состоит в выделении памяти для элемента данных вызывающего потока. После выделения памяти выполняется вызов процедуры *create\_window*, устанавливающей окно для потока и выделяющей память для переменной *win*, которая ссылается на окно. Наконец выполняется вызов *pthread\_setspecific()*, который связывает значение *win* с ключом. После этого, как только поток вызывает *pthread\_getspecific()*, передав глобальный ключ, он получает некоторое значение. Это значение было связано с этим ключом в вызывающем потоке, когда он вызвал функцию *pthread\_setspecific()*. Когда поток заканчивается, выполняются вызовы функций деструкторов, которые были настроены при вызове *pthread\_key\_create()*. Функция деструктора вызывается, если завершившийся поток установил значение для ключа вызовом *pthread\_setspecific()*.

Функция *pthread\_self()* вызывается для получения ID вызывающего ее потока:

*#include <pthread.h>*

*pthread\_t tid;*

*tid = pthread\_self();*

Функция *pthread\_equal()* вызывается для сравнения идентификаторов двух потоков:

*#include <pthread.h>*

*pthread\_t tid1, tid2;*

*int ret;*

*ret = pthread\_equal(tid1, tid2);*

Как и другие функции сравнения, *pthread\_equal()* возвращает значение, отличное от нуля, когда *tid1* и *tid2* равны; иначе возвращается 0. Если *tid1* или *tid2*- недействительный идентификатор потока, результат функции будет неопределенным.

Функция*pthread\_once()* используется для вызова процедуры инициализации потока только один раз. Последующие вызовы не оказывают никакого эффекта. Пример вызова функции:

*int pthread\_once(pthread\_once\_t \*once\_control, void (\*init\_routine)(void));*

Функция *sched\_yield()* приостанавливает текущий поток, чтобы процессор переключился на другой поток с тем же самым или большим приоритетом. Примервызова:

*#include <sched.h>*

*int ret;*

*ret = sched\_yield();*

После успешного завершения *sched\_yield()* возвращает 0. Если возвращается -1, то системная переменная*errno* устанавливается на код ошибки.

Функция *pthread\_setschedparam()* используется, чтобы изменить приоритет существующего потока. Эта функция никоим образом не влияет на дисциплину диспетчеризации:

*int pthread\_setschedparam(pthread\_t tid, int policy, const struct sched\_param \*param);*

Использование функции:

*#include <pthread.h>*

*pthread\_t tid;*

*int ret;*

*struct sched\_param param;*

*int priority;*

*/\* sched\_priority указывает приоритет потока \*/*

*sched\_param.sched\_priority = priority;*

*/\* единственный поддерживаемый алгоритм диспетчера\*/*

*policy = SCHED\_OTHER;*

*/\* параметры диспетчеризации требуемого потока \*/*

*ret = pthread\_setschedparam(tid, policy, &param);*

*pthread\_setschedparam()* возвращает 0 в случае успешного завершения, или другое значение в случае ошибки.

Функция:

*int pthread\_getschedparam(pthread\_t tid, int policy,  struct schedparam \*param);*

позволяет получить приоритет любого существующего потока.

Пример вызова функции:

*#include <pthread.h>*

*pthread\_t tid;*

*sched\_param param;*

*int priority;*

*int policy;*

*int ret;*

*/\* параметры диспетчеризации нужного потока \*/*

*ret = pthread\_getschedparam (tid, &policy, &param);*

*/\* sched\_priority содержит приоритет потока \*/*

*priority = param.sched\_priority;*

*pthread\_getschedparam()* возвращает 0 - в случае успешного завершения - или другое значение - в случае ошибки.

Поток, как и процесс, может принимать различные сигналы:

*#include <pthread.h>*

*#include <signal.h>*

*int sig;*

*pthread\_t tid;*

*int ret;*

*ret = pthread\_kill(tid, sig);*

*pthread\_kill()* посылает сигнал *sig* потоку, обозначенному *tid*, который должен быть потоком в пределах того же самого процесса, что и вызывающий поток. Аргумент *sig*должен быть действительным сигналом некоторого типа, определенного для функции *signal()* в файле < signal.h>.

Если *sig*имеет значение 0, выполняется проверка ошибок, но сигнал реально не посылается. Таким образом можно проверить правильность*tid.* Функция возвращает 0 - в случае успешного завершения - или другое значение - в случае ошибки.

Функция *pthread\_sigmask()* может использоваться для изменения или получения маски сигналов вызывающего потока:

*int pthread\_sigmask(int how, const sigset\_t \*new, sigset\_t \*old);*

Пример вызова функции:

*#include <pthread.h>*

*#include <signal.h>*

*int ret;*

*sigset\_t old, new;*

*/\* установка новой маски \*/*

*ret = pthread\_sigmask(SIG\_SETMASK, &new, &old);*

*/\* блокирование маски \*/*

*ret = pthread\_sigmask(SIG\_BLOCK, &new, &old);*

*/\* снятие блокировки \*/*

*ret = pthread\_sigmask(SIG\_UNBLOCK, &new, &old);*

*how* определяет режим смены маски. Он принимает значения следующих констант:

**SIG\_SETMASK** - заменяет текущую маску сигналов новой, при этом new указывает новую маску сигналов;

**SIG\_BLOCK** - добавляет новую маску сигналов к текущей, при этом new указывает множество блокируемых сигналов;

**SIG\_UNBLOCK** - удаляет new из текущей маски сигналов, при этом new указывает множество сигналов для снятия блокировки.

Если значение*new*равно NULL, то значение*how* не играет роли, и маска сигналов потока не изменяется. Чтобы узнать о блокированных в данный момент сигналах, аргумент *new* устанавливают в NULL. Переменная *old* указывает, где хранится прежняя маска сигналов, если ее значение не равно NULL.

Функция *pthread\_sigmask()* возвращает 0 - в случае успешного завершения - или другое значение - в случае ошибки.

### Остановка потока

Поток может прерваться несколькими способами. Первый способ предполагает возвращение управления из основной процедуры потока *start\_routine.* Второй способ - вызов *pthread\_exit()*, возвращающий статус выхода. Третий способ - прерывание потока с помощью функции *pthread\_cancel()*.

Функция *void pthread\_exit(void \*status)* прерывает выполнение потока точно так же, как функция*exit()* прерывает процесс:

*#include <pthread.h>*

*int status;*

*/\* выход возвращает статус status \*/*

*pthread\_exit(&status);*

Функция *pthread\_exit()* заканчивает выполнение вызвавшего ее потока. Все привязки данных для этого потока освобождаются. Если вызывающий поток не отделен, то ID этого потока и статус выхода (*status*) сохраняются, пока поток блокирован. В противном случае, статус игнорируется, а ID потока может быть немедленно использован для другого потока.

Функция*pthread\_cancel()* предназначена для прерывания потока:

*#include <pthread.h>*

*pthread\_t thread;*

*int ret;*

*ret = pthread\_cancel(thread);*

Способ обработки запроса на прерывание потока зависит от состояния указанного потока. Две функции, *pthread\_setcancelstate()* и *pthread\_setcanceltype()*, определяют это состояние; функция*pthread\_cancel()* возвращает 0 в случае успешного завершения, или другое значение в случае ошибки.

1. В базовом документе POSIX 1003.1 средства IPCSystemV не вводятся. [↑](#footnote-ref-1)
2. 2 Более точный порядок разрешения доступа к объекту IPC описан в спецификации SUSV2. [↑](#footnote-ref-2)