Министерство образования Республики Беларусь

Учреждение образования

Белорусский государственный университет информатики и радиоэлектроники

Факультет непрерывного и дистанционного обучения

Кафедра программного обеспечения информационных технологий

Электронный учебно-методический комплекс

по дисциплине

**ОПЕРАЦИОННЫЕ СИСТЕМЫИ СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ**

Часть 1

Для студентов специальности

1-40 01 01 Программное обеспечение информационных технологий

Минск 2010

# Общие сведения

## Сведения об ЭУМК

Электронный учебно-методический комплекспо дисциплине«Операционные системы и системное программирование»предназначен для студентов всех экономических и инженерно-экономических специальностей вузов, а также может быть использован преподавателями, аспирантами и практическими работниками предприятий.

Электронный учебно-методический комплекс составлен на основе рабочей учебной программы по курсу «Операционные системы», утверждённой деканом факультета непрерывного и дистанционного обучения \_\_\_\_.\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_.2010, регистрационный № УД 11‑\_\_\_\_\_\_\_/Р и рабочего учебного плана специальности 1-40 01 01«Программное обеспечение информационных технологий».

**Составители:**

**П.Ю. Бранцевич,**доцент кафедры программного обеспечения информационных технологий Учреждения образования «Белорусский государственный университет информатики и радиоэлектроники»,

**И.Г. Алексеев,** ассистент кафедры программного обеспечения информационных технологий Учреждения образования «Белорусский государственный университет информатики и радиоэлектроники».

Рассмотрен и рекомендован к изданию на заседании кафедры программного обеспечения информационных технологий, протокол протокол № \_16\_ от \_22.06.2009.

Одобрен и рекомендован к изданиюметодической комиссиейфакультетакомпьютерных систем и сетей, протокол № \_16\_ от \_22.06.2009.

## Методические рекомендации по изучению дисциплины

В соответствии с учебным планом студенты дистанционной формы обучения специальности«Программное обеспечение информационных технологий»изучают курс «Операционные системы и системное программирование».

Учебным планом по данному курсу предусмотрено изучение теоретических вопросов, практические задачи по наиболее актуальным темам, выполнение 2-х контрольных работ и 2-х индивидуальных практических работ (ИПР). Изучение курса заканчивается сдачей экзамена. К сдаче экзамена студенты допускаются только при условии выполненных и защищенныхИПР и контрольныхи работ.

Рекомендуется изучать курс «Операционные системы и системное программирование» в соответствии с рабочей программой. Сначала необходимо ознакомиться с содержанием курса, затем изучить рекомендуемую литературу, обращая внимание на вопросы, выделенные в рабочей программе, после чего изучить теоретическое изложение курса по приведенным разделам, темам и вопросам, ответить на контрольные вопросы, выполнитьконтрольные и ИПР работы в соответствии с индивидуальным заданием.

Теоретический материал базируется на практическом изучении ОС LINUX. Студенту необходимоустановить на свой компьютер одну из рекомендованных ОС LINUX ивыполнить на данной ОС контрольные работы и ИПР.

**Учреждение образования**

**«Белорусский государственный университет**

**информатики и радиоэлектроники»**

УТВЕРЖДАЮ

Декан факультета НиДО

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ В.М. Бондарик

\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_ 2010 г.

Регистрационный № УД-\_\_\_\_\_\_\_\_\_\_/р.

***ОПЕРАЦИОННЫЕ СИСТЕМЫ И СИСТЕМНОЕ***

***ПРОГРАММИРОВАНИЕ***

Часть 1

Рабочая учебная программа для специальности

1-40 01 01 Программное обеспечение информационных технологий

факультет Непрерывного и дистанционного обучения

кафедра Программное обеспечение информационных технологий

Курс 2\_

Часть 1\_

Индивидуальные практические Экзамен\_\_\_2\_\_(курс)

работы\_\_\_2\_\_(количество работ) Зачет\_\_\_\_\_\_\_\_\_\_\_(курс)

Курсовой проект

(работа)\_\_\_\_\_\_\_\_\_(курс)

Контрольные работы\_\_2\_\_(количество работ)

Всего часов Форма получения

по дисциплине\_640\_\_\_\_ высшего образования дистанционная

в том числе: первая часть \_226\_

2010

Составил

П.Ю. Бранцевич, доцент кафедры программного обеспечения информационных технологий Учреждения образования «Белорусский государственный университет информатики и радиоэлектроники» кандидат технических наук, доцент;

### Рабочая учебная программа составлена на основе типовой учебной программы «Операционные системы и системное программирование», утвержденной Министерством образования Республики Беларусь 03.06.2008 г., регистрационный № ТД – I.037/тип и учебного плана специальности 1- 40 01 01 «Программное обеспечение информационных технологий».

Рассмотрена и рекомендована к утверждению на заседании кафедры программного обеспечения информационных технологий

протокол № \_\_28\_ от \_\_31.05.2010\_\_

Заведующий кафедрой \_\_\_\_\_\_\_\_\_\_\_\_\_\_ В.В. Бахтизин

Одобрена и рекомендована к утверждению Научно-методической комиссией факультета «Компьютерные системы и сети» Учреждения образования «Белорусский государственный университет информатики и радиоэлектроники»

протокол № 16 от 14.06.2010

Председатель \_\_\_\_\_\_\_\_\_\_\_\_\_\_М.М. Лукашевич

Согласовано

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

Цель преподавания дисциплины

Целью курса является изучение: организации и принципов построения современных операционных систем и системных программ; аппаратных и программных средств современных процессоров, предназначенных для поддержки многозадачных операционных систем; методологии разработки системно-ориентированных программ с использованием современных алгоритмических языков и систем программирования.

Задачи изучения дисциплины

Задачи курса состоят в формировании базовых знаний в области: организации и функционирования современных операционных систем; способов разработки системного программного обеспечения с учетом аппаратно-программных особенностей вычислительной машины; современных систем программирования и разработки системного программного обеспечения.

В результате изучения первой части курса " Операционные системы и системное программирование" обучаемый должен:

знать:

* определение и функции операционной системы;
* принципы построенияи работы операционных систем;
* типовые алгоритмы организации взаимодействия между процессами;
* подсистемы ОС;
* аппаратные и программные средства современных процессоров, предназначенных для поддержки многозадачных операционных систем;
* задачи, решаемые при управлении виртуальной памятью;
* современные операционные системы;
* основные системные вызовы стандарта POSIX;
* систему прав доступа в файловой системе ОС семейства UNIX/Linux;
* способы взаимодействия между процессами в ОС семейства UNIX/Linux;

уметь:

* разрабатывать программы, в операционной системе UNIX/Linux;
* разрабатывать программы, по организации взаимодействия между процессами в ОС UNIX;
* разрабатывать многопоточные программы с синхронизацией данных.

Для изучения курса «Операционные системы и системное программирование» необходимы знания принципов работы ЭВМ, навыки алгоритмизации и разработки программ на алгоритмических языках высокого уровня и машинно-ориентированных языках, работы с операционными системами и системами программирования на уровне, достаточном для отладки и выполнения программ. Курс базируется на сведениях, полученных при изучении курсов "Основы алгоритмизации и программирования", "Основы компьютерной техники", «Языки программирования».

**СОДЕРЖАНИЕ ДИСЦИПЛИНЫ**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Ку  р  с | Название и содержание тем (*по типовой или учебной программе*) | Контрольная работа (номер и тема по п.2) | Лабораторная работа | Индивидуальная практическая работа (по п.12) | Оснащение контрольных, лабораторных и индивидуальных практических работ  (по п.4) | Литература (по п.3 | Рекомендуемый объем для изучения (в часах)3 | Форма контроля знаний (зачет по контрольной работе, тесты, защита лабораторной работы, защита курсового проекта, экзамен, зачет) |
| 1 | 2 | 3 | 4 | | 5 | 6 | 7 | 8 |
| Раздел 1. Операционная система UNIX (LINUX) | | | | | | | | |
| 2 | **Тема1. Разработка программ в ОС UNIX**  Стандарты UNIX. Пользователи системы, атрибуты пользователя. Создание программы, исходный текст, заголовки. Системные вызовы и функции стандартных библиотек. Обработка ошибок. Форматы выполняемых файлов. |  |  | 1 | 4.1  4.2  4.3 | 3.1.3  3.1.4  3.1.7  3.1.12 | 8+20=28 | Зачет по индивидуальной практической работе |
| 2 | **Тема 2. Файловая система ОС UNIX**  Типы файлов. Владельцы файлов. Управление правами доступа в файловой системе. Атрибуты файлов. Управление свойствами файлов. Работа с файлами. Структура файловой системы. |  |  | 2 | 4.1  4.2  4.3 | 3.1.3  3.1.4  3.1.11 | 16+20=  36 | Зачет по индивидуальной практической работе. Тест 1 |
| 2 | **Тема 3. Процессы в ОС UNIX**  Понятие процесса. Типы процессов. Атрибуты процессов. Создание и управление процессами. Перегрузка процессов. Завершение процессов. Синхронизация процессов. Системные средства для управления свойствами процессов. | 1 |  |  | 4.1  4.2  4.3 | 3.1.3  3.1.8  3.1.10  3.1.11 | 16+15=  31 | Зачет по контрольной работе |
| 2 | **Тема 4. Взаимодействие между процессами**  Сигналы. Обработка сигналов. Неименованные каналы. Именованные каналы. Дополнительные средства взаимодействия между процессами. Сообщения, семафоры, разделяемая память. Понятие потока в ОС UNIX. | 2 |  |  | 4.1  4.2  4.3 | 3.1.3  3.1.8  3.1.10  3.1.11 | 20+25=  45 | Зачет по контрольной работе.  Тест 2 |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| Раздел 2. Операционные системы | | | | | | | | |
| 2 | **Тема 5. Введение в операционные системы**  Назначение, основные этапы развития операционных систем. Особенности современных операционных систем. Принципы построения ОС. |  |  |  |  | 3.1.1  3.1.2 | 8 | Ответы на вопросы для  самоконтроля |
| 2 | **Тема 5. Основы операционных систем**  Понятие процесса, потока, ресурса, свойства, классификация. Концепция виртуализации. Концепция прерывания. |  |  |  |  | 3.1.1  3.1.2 | 8 | Ответы на вопросы для самоконтроля |
| 2 | **Тема 7. Процессы**  Состояния процессов. Описание процессов. Взаимодействие процессов. Задача взаимного исключения. Решение задачи взаимного исключения. Алгоритм Деккера. Алгоритм Петерсона. Семафоры. Применение семафоров для решения задачи взаимного исключения. Задача производители-потребители и ее решения. Мониторы. Задача «читатели-писатели». Процессы и потоки. | 1,2 |  |  | 4.1  4.2  4.3 | 3.1.1  3.1.2 | 16+10=26 | Ответы на вопросы для  самоконтроля.  Зачет по контрольной работе |
| 2 | **Тема 8. Ресурсы**  Распределение ресурсов, проблема тупиков. Алгоритм банкира. Применение алгоритма банкира. |  |  |  |  | 3.1.1  3.1.2 | 4 | Ответы на вопросы для самоконтроля |
| 2 | **Тема 9. Память. Управление памятью**  Требования к управлению памятью. Схемы распределения памяти. Страничная организация памяти. Сегментация памяти. |  |  |  |  | 3.1.1  3.1.2 | 8 | Ответы на вопросы для самоконтроля |
| 2 | **Тема 10. Организация виртуальной памяти**  Структуризация адресного пространства виртуальной памяти. Задачи управления виртуальной памятью: задача размещения, задача перемещения, задача преобразования адресов, задача замещения. |  |  |  |  | 3.1.1  3.1.2 | 8 | Ответы на вопросы для  самоконтроля.  Тест 3 |
| 2 | **Тема 11. Планирование в операционных системах**  Типы планирования. Алгоритмы планирования. Примеры реализации алгоритмов планирования в современных операционных системах |  |  |  |  | 3.1.1  3.1.2 | 4 | Ответы на вопросы для  самоконтроля |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 2 | **Тема 12. Управление вводом-выводом и файлами**  Организация функций ввода-вывода. Логическая структура функций ввода-вывода. Буферизация операций ввода-вывода. Дисковое планирование. Система управления файлами. Организация файлов, доступ к файлам. Управление внешней памятью. |  |  |  |  | 3.1.1  3.1.2 | 4 | Ответы на вопросы для  самоконтроля. Зачет по индивидуальной практической работе. |
| 2 | **Тема 13. Аппаратно-программные особенности современных процессоров, ориентированные на поддержку многозадачных систем**  Управление памятью в реальном и защищенном режимах. Дескрипторные таблицы и дескрипторы сегментов. Организация защиты по данным и кодам. Команды и структуры данных для управления в многозадачных режимах. Организация прерываний. |  |  |  |  | 3.1.1  3.1.2  3.2.6  3.2.7 | 16 | Ответы на вопросы для  самоконтроля.  Тест 3 |
|  | | | | | | | | Экзамен |

1 – лабораторные работы выполняются студентами очно в учебной лаборатории университета в составе учебной подгруппы ,

2 – индивидуальные практические работы выполняются студентами дистанционно вне стен учебной лаборатории на базе информационно0коммуникационных технологий

1. ИНДИВИДУАЛЬНЫЕ ПРАКТИЧЕСКИЕ РАБОТЫ

И ИХ ХАРАКТЕРИСТИКА

|  |  |  |  |
| --- | --- | --- | --- |
| № пп | Тема | Характеристика | Рекомендуемый объем в часах |
| 1. | Система команд, файловая структура,работа с файлами иуправление ОС Unix/Linu*x* с помощью интерпретатора BASH | Цель работы – изучение команд ОС UNIX для работы с файлами, каталогами, дисками, системной датой и временем; текстового редактора и файлового менеджера, исследовать основные объекты, команды, типы данных и операторы управления интерпретатора bash; создать скрипт-файл.  Пример задания. Написать программу копирования одного файла в другой, имена файлов передаются в качестве аргументов командной строки при запуске программы. Предусмотреть копирование прав доступа к файлу. | 20 |
| 2. | Основные принципы программирования, процессы и потоки в ОС Unix/Linux | Цель работы – изучение файловой системы ОС Linux и основных функций для работы с каталогами и файлами, исследовать методы создания процессов в ОС Linux, основные функции создания и управления процессами, обмен данными между процессами.  Пример задания. Разработать программу «интерпретатор команд», которая воспринимает команды, вводимые с клавиатуры, и осуществляет их корректное выполнение. Для этого каждая вводимая команда должна выполняться в отдельно запускаемом процессе с использованием вызова *exec()*. Предусмотреть контроль ошибок. | 20 |

##### 2. КОНТРОЛЬНЫЕ РАБОТЫ, ИХ ХАРАКТЕРИСТИКА

|  |  |  |  |
| --- | --- | --- | --- |
| № пп | Тема | Характеристика | Рекомендуемый объем в часах |
| 1 | 2 | 3 | 4 |
| 1. | Средства межпроцессного взаимодействия (сигналы) | Цель работы – изучить методы и средства взаимодействия процессов с использованием сигналов в ОС Linux.  Пример задания. Организовать функционирование процессов следующей структуры: отец-сын-сын.  Процессы определяют свою работу выводом сообщений вида :  ***N pid ppid текущее время*** (мсек) (***N*** – текущий номер сообщения) на экран. “Отец” одновременно, посылает сигнал ***SIGUSR1*** “сыновьям”. “Сыновья” получив данный сигнал, посылают в ответ “Отцу” сигнал ***SIGUSR2***. “Отец” получив сигнал ***SIGUSR2***, через время ***t=100*** мсек одновременно, посылает сигнал ***SIGUSR1*** “сыновьям”. И так далее… Написать функции-обработчики сигналов, которые при получении сигнала выводят сообщение о получении сигнала на экран. При получении/посылке сигнала они выводят соответствующее сообщение:  ***N pid ppid текущее время (мсек***) сын такой-то get/put SIGUSRm.  Предусмотреть механизм для определения “Отцом”, от кого из “Сыновей” получен сигнал. | 15 |
| 1 | 2 | 3 | 4 |
| 2. | Средства межпроцессного взаимодействия (каналы, разделяемая память, семафоры) | Цель работы – изучить методы и средства взаимодействия процессов с использованием каналов, разделяемой памяти и семафоров в ОС Linux.  Пример задания. Cоздать два дочерних процесса. Родительский процесс создаёт семафоры (*сем1*), (*сем2*) и *2* неименованных канала (*кан1 и кан2)*. Оба дочерних процесса непрерывно записывают в каналы по *100* строк вида: *номер\_строкиpid\_процессатекущее\_время* (мсек). Всего процессы должны записать *1000* строк. Семафоры (*сем1*), (*сем2*) используются процессами для разрешения кому из процессов получить доступ к каналу. Родительский процесс читает из каждого канала по *75* строк и выводит их на экран. Дочерние процессы начинают операции с каналами после получения сигнала *SIGUSR2* от родительского процесса. | 25 |

##### 3 УЧЕБНО-МЕТОДИЧЕСКИЕ МАТЕРИАЛЫ ПО ДИСЦИПЛИНЕ

##### 3.1 Основная литература

3.1.1. Гордеев А.В., Молчанов А.Ю. Системное программное обеспечение. - СПб.: Питер, 2002. -736 с.

3.1.2. Столлингс В. Операционные системы. - М.: Издательский дом "Вильямс", 2002. -848 с.

3.1.3. Хэвиленд К., Грэй Д., Салама Б. Системное программирование в UNIX. Руководство программиста по разработке ПО. -М.: ДМК Пресс, 2000. -368 с.

3.1.4. Робачевский А.М. Операционная система UNIX. -СПб.: BHV - Санкт-Петербург, 1997. -528 с.

3.1.5. Стивенс У. UNIX: взаимодействие процессов. - Спб.: Питер, 2002. - 576 с.

3.1.6. Стивенс У. UNIX: разработка сетевых приложений. Спб.: Питер, 2003. - 1088 с.

3.1.7. Моли Б. Unix/Linux: теория и практика программирования. - М.: КУДИЦ-ОБРАЗ, 2004, -576 с.

3.1.8. Роббинс А. Linux: программирование в примерах. - М.: КУДИЦ-ОБРАЗ, 2005, -656 с.

3.1.9. Петерсен Р. LINUX: руководство по операционной системе: Пер. с англ. - К.: Издательская группа BHV, 1997. - 688 с.

3.1.10. Чан Т. Системное программирование на С++ для UNIX: Пер. с англ. - К.: Издательская группа BHV, 1997. - 592 с.

3.1.11. Лав Р. Linux. Системное программирование. – СПб.: Питер, 2008. – 416 с.

3.1.12. Шойхер М., Тоботрас Б. Как установить Linux и работать с ним. – М.: ООО МЦФ, 1998. -80 с.

##### 3.2 Дополнительная литература

3.2.1. Дейтел Р. Введение в операционные системы.1,2т. -М.:Мир, 1987.

3.2.2. Кейслер С. Проектирование операционных систем для малых ЭВМ: Пер. с англ. -М.: Мир, 1986. - 680 с.

3.2.3. Немет Э., Снайдер Г., Сибасс С., Хейн Т.Р. UNIX: руководство системного администратора : Пер. с англ. - К.: BHV, 1996. - 832 с.

3.2.4. Керниган Б.В., Пайк Р. UNIX - универсальная среда программирования: Пер. с англ., - М.: Финансы и статистика, 1992. - 304 с.

3.2.5. Краковяк С. Основы организации и функционирования ОС ЭВМ. -М.: Мир, 1988.

3.2.6. Григорьев В.Л. Микропроцессор i486. Архитектура и программирование (в 4-х книгах). Книга 1. Программная архитектура. - М.: ГРАНАЛ, 1993. -346 с.

3.2.7. Григорьев В.Л. Микропроцессор i486. Архитектура и программирование (в 4-х книгах). Книга 2. Аппаратная архитектура. Книга 3. Устройство с плавающей точкой. Книга 4. Справочник по системе команд. - М.: ГРАНАЛ, 1993. -382 с.

3.1.8. Бек Л. Введение в системное программирование. -М: Мир, 1988.

4 ПЕРЕЧЕНЬ КОМПЬЮТЕРНЫХ ПРОГРАММ, НАГЛЯДНЫХ

И ДРУГИХ ПОСОБИЙ, МЕТОДИЧЕСКИХ УКАЗАНИЙ

И МАТЕРИАЛОВ К ТЕХНИЧЕСКИМ СРЕДСТВАМ ОБУЧЕНИЯ

4.1. Операционная система UNIX(Linux), например, RedHad, Mandreke с текстовым редактором для набора программ на языке Си.

4.2. Алексеев И.Г., Бранцевич П.Ю. Теория вычислительных процессов и структур: Учебно-метод. пособие для студ. спец. «Программное обеспечение информационных технологий» дневной формы обуч. Мн.: БГУИР, 2004. -52с.

4.3. УМК по предмету «Операционные системы и системное программирование», Часть 1.

**Протокол согласования рабочей программы**

**с другими дисциплинами специальности**

|  |  |  |  |
| --- | --- | --- | --- |
| Название дисциплины, изучение которой связано с дисциплиной рабочей  программы | Кафедра, обеспечивающая изучение этой дисциплины | Предложения кафедры об изменениях в содержании рабочей программы | Решение, принятое кафедрой, разработавшей рабочую программу (с указанием даты и номера протокола) |
| 1 | 2 | 3 | 4 |
| Компьютерные системы и сети | ПОИТ | Принять без изменений | Принять без изменений  Протокол № 28  От 31.05.2010 |

Зав. кафедрой ПОИТ В.В. Бахтизин

**1. ИНДИВИДУАЛЬНЫЕ ПРАКТИЧЕСКИЕ РАБОТЫ**

**И ИХ ХАРАКТЕРИСТИКА**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № пп | Тема | Характеристика | Форма проведения | Рекомендуемый объем в часах |
| 1. | Изучение UNIX-подобных ОС | 1. Цель работы – изучение файловой системы ОС Linux и основных функций для работы с каталогами и файлами. Примерный вариант задания: найти в заданном каталоге и всех его подкаталогах все файлы, заданного размера. Имя каталога задаётся пользователем в качестве третьего аргумента командной строки. Диапазон мин. мах. размеров файлов задаётся пользователем в качестве первого и второго аргумента командной строки. Имя владельца и каталог задаются пользователем в качестве первого и второго аргумента командной строки. Программа выводит результаты поиска в файл (четвертый аргумент командной строки) в виде полный путь, имя файла, его размер. На консоль выводится общее число просмотренных файлов. Варианты задания выбираются из методического пособия по лабораторным работам в составе ЭУМК | виртуально | 4 |
| 2. | Исследовать методы создания процессов в ОС Linux/UNIX, основные функции создания и управления процессами, обмен данными между процессами. | Цель работы: изучение основных методов создания и управления процессами (потоками) и способов их взаимодействия в среде ОС Linux/UNIX. Примерный вариант задания: Создать дерево процессов(потоков) и организовать обмен информацией между конкретными процессами(потоками) посредством общего файла. Варианты задания выбираются из методического пособия по лабораторным работам в составе ЭУМК | виртуально | 4 |

**2. КОНТРОЛЬНЫЕ РАБОТЫ, ИХ ХАРАКТЕРИСТИКА**

|  |  |  |  |
| --- | --- | --- | --- |
| № пп | Тема | Характеристика | Рекомендуемый объем в часах |
|  | Управление ОС Linux/UNIX с помощью интерпретатора BASH | Цель работы: изучение файловой структуры ОС, файловых менеджеров (Midnight Commander). Команды ОС для работы с файлами, каталогами, дисками, системной датой и временем, команды интерпретатора BASH. Примерный вариант задания:Написать скрипт находящий в заданном каталоге и всех его подкаталогах все файлы, владельцем которых является заданный пользователь. Имя владельца и каталог задаются пользователем в качестве первого и второго аргумента командной строки. Скрипт выводит результаты в файл (третий аргумент командной строки) в виде полный путь, имя файла, его размер. На консоль выводится общее число просмотренных файлов. Варианты задания выбираются из методического пособия по лабораторным работам в составе ЭУМК | 8 |
| 2. | Методы и средства межпроцессного взаимодействия ОС Linux/UNIX | Цель работы: изучить методы и средства взаимодействия процессов в ОС Linux/UNIX. Примерный вариант задания: Cоздать два дочерних процесса. Родительский процесс создаёт семафор (сем1) и общий файл отображенный в память. Оба дочерних процесса непрерывно записывают в файл по 100 строк вида: номер\_строки pid\_процесса текущее\_время (мсек). Всего процессы должны записать 1000 строк. Семафор сем1 используется процессами для разрешения кому из процессов получить доступ к файлу. Родительский процесс читает из файла по 75 строк и выводит их на экран. Дочерние процессы начинают операции с файлом после получения сигнала SIGUSR1 от родительского процесса. Варианты задания выбираются из методического пособия по лабораторным работам в составе ЭУМК. | 8 |

3 УЧЕБНО-МЕТОДИЧЕСКИЕ МАТЕРИАЛЫ ПО ДИСЦИПЛИНЕ

3.1 Основная литература

3.1.1. Гордеев А.В., Молчанов А.Ю. Системное программное обеспечение. - СПб.: Питер, 2002. -736 с.

3.1.2. Столлингс В. Операционные системы. - М.: Издательский дом "Вильямс", 2002. -848 с.

3.1.3. Хэвиленд К., Грэй Д., Салама Б. Системное программирование в UNIX. Руководство программиста по разработке ПО. -М.: ДМК Пресс, 2000. -368 с.

3.1.4. Робачевский А.М. Операционная система UNIX. -СПб.: BHV - Санкт-Петербург, 1997. -528 с.

3.1.5. Стивенс У. UNIX: взаимодействие процессов. - Спб.: Питер, 2002. - 576 с.

3.1.6. Стивенс У. UNIX: разработка сетевых приложений. Спб.: Питер, 2003. - 1088 с.

3.1.7. Моли Б. Unix/Linux: теория и практика программирования. - М.: КУДИЦ-ОБРАЗ, 2004, -576 с.

3.1.8. Роббинс А. Linux: программирование в примерах. - М.: КУДИЦ-ОБРАЗ, 2005, -656 с.

3.1.9. Петерсен Р. LINUX: руководство по операционной системе: Пер. с англ. - К.: Издательская группа BHV, 1997. - 688 с.

3.1.10. Чан Т. Системное программирование на С++ для UNIX: Пер. с англ. - К.: Издательская группа BHV, 1997. - 592 с.

3.1.11. Лав Р. Linux. Системное программирование. – СПб.: Питер, 2008. – 416с.

3.1.12. Шойхер М., Тоботрас Б. Как установить Linux и работать с ним. – М.: ООО МЦФ, 1998. -80 с.

3.2 Дополнительная литература

3.2.1. Дейтел Р. Введение в операционные системы.1,2т. -М.:Мир, 1987.

3.2.2. Кейслер С. Проектирование операционных систем для малых ЭВМ: Пер. с англ. -М.: Мир, 1986. - 680 с.

3.2.3. Немет Э., Снайдер Г., Сибасс С., Хейн Т.Р. UNIX: руководство системного администратора : Пер. с англ. - К.: BHV, 1996. - 832 с.

3.2.4. Керниган Б.В., Пайк Р. UNIX - универсальная среда программирования: Пер. с англ., - М.: Финансы и статистика, 1992. - 304 с.

3.2.5. Краковяк С. Основы организации и функционирования ОС ЭВМ. -М.: Мир, 1988.

3.2.6. Григорьев В.Л. Микропроцессор i486. Архитектура и программирование (в 4-х книгах). Книга 1. Программная архитектура. - М.: ГРАНАЛ, 1993. -346 с.

3.2.7. Григорьев В.Л. Микропроцессор i486. Архитектура и программирование (в 4-х книгах). Книга 2. Аппаратная архитектура. Книга 3. Устройство с плавающей точкой. Книга 4. Справочник по системе команд. - М.: ГРАНАЛ, 1993. -382 с.

3.1.8. Бек Л. Введение в системное программирование. -М: Мир, 1988.

4 ПЕРЕЧЕНЬ КОМПЬЮТЕРНЫХ ПРОГРАММ, НАГЛЯДНЫХ

И ДРУГИХ ПОСОБИЙ, МЕТОДИЧЕСКИХ УКАЗАНИЙ

И МАТЕРИАЛОВ К ТЕХНИЧЕСКИМ СРЕДСТВАМ ОБУЧЕНИЯ

4.1. Операционная система UNIX(Linux), например, SUSE , RedHad, Mandrake с текстовым редактором для набора программ на языке Си.

4.2. Алексеев И.Г., Бранцевич П.Ю. Теория вычислительных процессов и структур: Учебно-метод. пособие для студ. спец. «Программное обеспечение информационных технологий» дневной формы обуч. Мн.: БГУИР, 2004. -52с.

4.3. УМК по предмету «Операционные системы и системное программирование», Часть 1.

# Теоретический раздел

## Лекции

## 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**

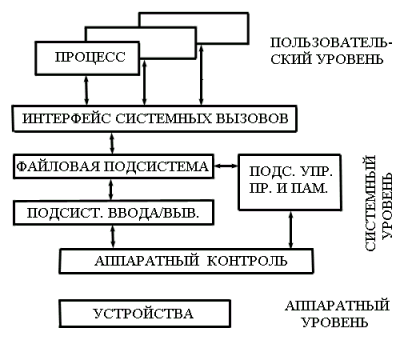


Рис 1.2. Ядро системы

**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 ⎯ 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**.

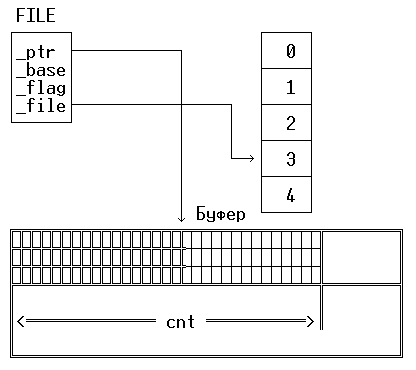
2. Построчная буферизация. Практический ввод/вывод выполняется с помощью системных вызовов **read**, **write** построчно при обнаружении конца строки (перевода каретки). Обычно применяется для стандартных потоков ввода/вывода.

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:

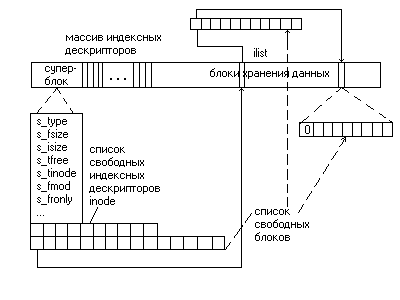


Рис. 2.6. Структура файловой системы s5fs

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:

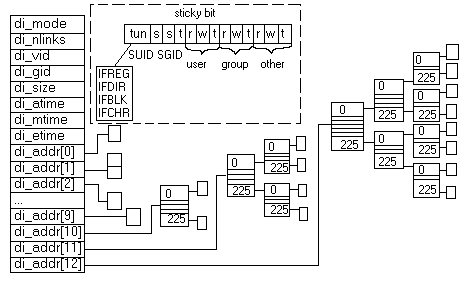


Рис. 2.7. Структура индексного дескриптора

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

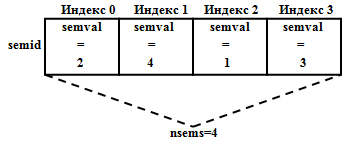


Рис. 4.1. Набор семафоров

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.*

**Понятие потока в ОС UNIX**

.1 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.

.2 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.

.2 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.

.3 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.

.4 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];*

*}*

.5 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.

.6 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);*

Кактолькоключудален, любаяссылкананегочерез*pthread\_setspecific()*или*pthread\_getspecific()*приводиткошибкеEINVAL. 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.

***Тема 5.Введение в операционные системы***

**Понятие операционной системы**

***Операционная система*** – это набор программ, которые обеспечивают возможность использования аппаратуры компьютера. При этом аппаратные средства представляют собой некоторую вычислительную мощность, а задача операционной системы заключается в том, чтобы сделать аппаратные средства доступными и по возможности удобными для пользователя.

ОС реализует следующие основные функции:

- определяет или обеспечивает интерфейс пользователя (интерфейс – унифицированная система связи между пользователем и машиной);

- обеспечивает разделение аппаратных ресурсов между пользователями;

- дает возможность работать с общими данными в режиме коллективного пользования;

- планирует доступ пользователей к общим ресурсам;

- обеспечивает эффективное выполнение операций ввода-вывода;

- осуществляет восстановление информации и вычислительного процесса в случае ошибки.

ОС управляет следующими основными ресурсами:

- процессорами;

- памятью;

- устройствами ввода-вывода;

- данными.

ОС взаимодействует с:

- операторами ЭВМ;

- прикладными программистами;

- системными программистами;

- административным персоналом;

- пользователями;

- аппаратными средствами и программами.

***Пользователи*** – это абоненты вычислительного комплекса, которые применяют компьютер для выполнения полезной работы.

***Операторы ЭВМ*** – это специально подготовленные люди, которые следят за работой ОС и по запросам системы вмешиваются в работу компьютера для устранения каких-либо препятствий или выполнения необходимых действий.

***Системные программисты*** занимаются сопровождением ОС, осуществляют ее настройку применительно к требованиям конкретной машины, решаемых задач и эксплуатируемых условий. При необходимости осуществляют разработку дополнительных системных программ, которые направлены на расширение функциональных возможностей системы и для обслуживания новых аппаратных устройств.

***Администраторы*** – это люди, устанавливающие принципы и порядок работы на ЭВМ и взаимодействующие с ОС, чтобы обеспечить соблюдение принятого порядка.

**Поколения ОС**

Выделяют следующие поколения ОС:

***Нулевое поколение*** (40-е гг. ХХ в). Характеризуется тем, что ОС на первых вычислительных машинах не было. Пользователи имели доступ к машинному языку и все программы писались на машинных кодах.

***Первое поколение*** (50-е гг. ХХ в). ОС на этом этапе были разработаны с целью ускорения и упрощения перехода с задачи на задачу. Это было началом систем пакетной обработки. Пакетная обработка предусматривает объединение отдельных задач в группы. Каждая задача на этапе выполнения получает все ресурсы машины, а после завершения задачи(нормального или аварийного), управление ресурсами возвращалось ОС, которая очищала машину от последней решенной задачи и обеспечивала ввод и запуск следующей задачи. Считается, что первую ОС в начале 50-х гг. для IBM 701 создала исследовательская лаборатория фирмы GeneralMotors.

Первые ОС ориентировались на сокращение времени, которое затрачивалось на запуск задачи на вычислительной машине и на удаление ее из машины. Т.е. цель – минимизация времени перехода с одной задачи на другую.

Уже в первых ОС появилась концепция имен системных файлов как средство достижения независимости программ от аппаратуры. Это позволило указывать программе не конкретный номер физического устройства, а стандартные системные файлы ввода-вывода.

К концу 50-х гг. были разработаны ОС, обладающие следующими характеристиками:

- пакетная обработка одного потока задач;

- наличие стандартных подпрограмм ввода-вывода. С целью освобождения пользователя от программирования процессов ввода-вывода на машинном языке;

- возможность автоматического перехода от программы к программе;

- наличие средств восстановления после ошибок, которые позволяли запускать следующую задачу при минимальном вмешательстве операторов;

- наличие языков управления заданиями, с помощью которых пользователи описывали свои задания и ресурсы, требуемые для их выполнения.

***Второе поколение*** (первая половина 60-х гг.). Особенностью ОС этого поколения явилось то, что они создавались как системы коллективного пользования с мультипрограммным режимом работы и как первые системы мультипроцессорного типа.

В мультипрограммных системах несколько пользовательских программ одновременно находятся в оперативной памяти компьютера, а центральный процессор быстро переключается с задачи на задачу. Мультипроцессорные системы содержат несколько процессоров с целью повышения вычислительной мощности. В это время разработаны методы, обеспечивающие независимость программирования от внешних устройств, что привело к тому, что пользователь указывал не конкретное физическое устройство ввода-вывода, а определял характеристики, которым должно отвечать устройство ввода-вывода, а ОС сама находила соответствующее устройство и при необходимости давала оператору указание подготовить это устройство к работе.

На этом этапе разработаны первые системы с разделением времени, которые позволяли пользователю непосредственно взаимодействовать с компьютером при помощи пультов-терминалов телетайпного типа. С системой разделения времени пользователи работают в диалоговом или интерактивном режимах. Это позволило значительно повысить эффективность разработки программ. Наиболее распространенная система такого типа – программа SABRE бронирования и продажи билетов на самолеты компании AmericanAirlines.

Чуть позже появились первые системы реального времени, в которых компьютеры применялись для управления технологическими процессами. Для систем реального времени характерно то, что они обеспечивают реакцию на предусмотренные события за время, не превышающее некоторое допустимое для данного события.

***Третье поколение*** (середина 60-х – конец 70-х гг.). Это поколение связано с созданием и развитием больших универсальных машин System360/370 фирмы IBM. Эти машины относились к кассу машин общего назначения. Но в СССР был разработан аналог машин - ЕС.

ОС третьего поколения стали многорежимными системами, т.е. некоторые из них обеспечивали работу практически во всех режимах(пакетная обработка, режим разделения времени, режим реального времени, мультипроцессорный режим).Такие ОС были громоздкими и дорогостоящими. Сроки и затраты на их реализацию значительно превышали планирование.

ОС стали толстой прослойкой между пользователем и вычислительной машиной. Пользователю приходилось изучать сложные языки управления заданиями, чтобы уметь описать задание и требуемые для их выполнения ресурсы.

***Четвертое поколение*** (конец 70-х – начало 90-х гг.). Этот этап имеет следующие особенности:

- широкое распространение вычислительных сетей и средств обработки данных в режиме on-line, т.е. пользователи получают доступ к территориально распределенным компьютерам при помощи терминалов различного типа;

- появление микропроцессоров создало условия микрокомпьютера, который привел к существенным социальным последствиям;

- потребность передачи информации по линиям связи различных типов потребовала создания систем защиты данных, т.е. шифрованию, созданию ключей защиты, разделению прав доступа к информации;

- большое внимание стало уделяться созданию ОС, ориентированных на неподготовленного пользователя, созданию дружественных пользовательских интерфейсов.

В это время появились системы с управлением при помощи меню. Начала широко распространяться концепция виртуальных машин. Важную роль стали играть системы баз данных (БД). Получила распространение концепция о распределенной обработке данных. Классический пример ОС этого поколения – ОС UNIX.

***Пятое поколение*** (начало 90-х гг.). Отличительные особенности этого этапа следующие.

1. Интенсивное развитие и широкое распространение и развитие персонального компьютера.

2. Переход компьютера из сферы производства в бытовую сферу.

3. Развитие локальных и глобальных сетей, распространение Internetа.

4. Рост вычислительной и информационной мощности.

5. Решение нетрадиционных задач для вычислительной техники.

6. Совершенствование пользовательского интерфейса.

7. Развитие графических пользовательских интерфейсов.

8. Появление и развитие новых устройств для взаимодействия пользователя и вычислительной машины (манипуляторы, экраны управления, цифровые камеры, речевые вводы и др.).

9. Развитие интеллектуальных систем и систем искусственного интеллекта.

10. Создание супер-ЭВМ, обладающих огромной вычислительной мощностью.

Получили развитие многозадачные ОС.

**Функции и свойства ОС**

***Операционная система (ОС)*** – это упорядоченная последовательность системных управляющих программ, совместно с необходимыми информационными массивами, предназначенных для планирования, исполнения пользовательских программ и управления всеми ресурсами вычислительной машины (программами, данными, аппаратурой и другими распределяемыми и управляемыми объектами) с целью предоставления возможности пользователям эффективно, в некотором смысле, решать задачи, сформулированные в терминах вычислительной машины.

ОС – это программа, которая контролирует работу прикладных программ и системных приложений и выполняет роль интерфейса между приложениями и аппаратным обеспечением ЭВМ (это более простое определение).

Можно определить следующие параметры ОС:

1. Удобство (ОС делает использование ПК достаточно простым и удобным)
2. Эффективность (ОС позволяет эффективно использовать ресурсы ПК)
3. Возможность развития (ОС должна быть организована так, чтобы допускать эффективную разработку, тестирование и внедрение новых приложений и системных функций, причем это не должно мешать нормальному функционированию ОС).

Рассматривая ОС как интерфейс между пользователем и компьютером, можно представить иерархическую структуру программного и аппаратного обеспечения, использующегося для представления конечному пользователю возможности работы с приложениями.

Аппаратное обеспечение

**ОС**

Утилиты

Прикладные программы

Конечные пользователи

Программисты

Разработчики ОС

Рис. 1.1. Место операционной системы в вычислительной системе

ОС предоставляют следующий перечень услуг:

1. Разработка программ. ОС предоставляет программисту разнообразные инструменты и сервисы, к примеру, редакторы, отладчики, реализованные в виде программ-утилит, которые поддерживаются ОС, хотя не входят в её ядро. Такие программы называются *инструментами разработки приложений.*
2. Исполнение программ. Для запуска программы требуется загрузить её в основную память (сформировать области команд и данных), инициализировать устройства ввода-вывода и файлы, подготовить ресурсы ЭВМ. ОС выполняет эти трудоемкие операции. Существуют специальные загрузчики программ для выполнения этих действий.
3. Доступ к устройствам ввода-вывода. Несмотря на то, что для каждого устройства нужен свой набор команд и контроллеров сигналов, ОС предоставляет пользователю единообразный интерфейс, который скрывает эти детали и обеспечивает программисту доступ к устройствам ввода-вывода с помощью простых команд чтения-записи.
4. Контролируемый доступ к файлам. При работе с файлами обеспечивается необходимая структуризация данных, записываемая в файлы, а также работа механизма защиты при обращении к файлу многопользовательскими ОС (классический пример *–* ОС UNIX).
5. Системный доступ. ОС управляет доступом к отдельным системным ресурсам, а также доступом к вычислительной системе в целом. Она защищает ресурсы и данные от несанкционированного использования и разрешает конфликтные ситуации (например, проблема тупиков).
6. Обнаружение ошибок и их обработка. В ходе работы ЭВМ возможны сбои, например ошибки памяти, нарушение работы отдельных устройств, возможные программные ошибки (переполнения, попытки обращения к недоступным ресурсам). В этих случаях ОС должна выполнять действия, минимизирующие влияние ошибки на работу приложения. Спектр таких действий достаточно широк, от простого уведомления об ошибке до аварийной остановки программы.
7. Учет использования ресурсов. В ОС должны быть средства учета использованных ресурсов и отображения параметров их производительности, а также определение времени использования ресурса отдельными пользователями. Это особенно важно при настройке конфигурации. Наличие таких средств позволяет оптимизировать работу системы в целом и обеспечить наиболее оптимальную загрузку процессора и других ресурсов, т. е. повысить ее производительность.

**Характеристики современных ОС**

В экспериментальных коммерческих ОС были опробованы различные подходы и структурные элементы, большинство из которых можно объединить в следующие категории:

– архитектура микроядра;

– многопоточность;

– симметричная многопроцессорность;

– распределенные ОС;

– объектно-ориентированный дизайн.

**Архитектура микроядра**

В настоящее время многие ОС реализованы по принципу большого монолитного ядра. Такое ядро обеспечивает планирование выполнения программ, работу с файловой системой, сетевые функции, работу драйверов различных устройств, управление памятью и многие другие функции.

Монолитное ядро реализуется как единый процесс, все элементы которого реализуют одно и то же адресное пространство.

В архитектуре микроядра к функциям ядра относятся только самые важные, такие как работа с адресным пространством, обеспечения взаимодействия между процессами, основное планирование. Обеспечение остальных услуг ОС возможно на процессы, которые запускаются в пользовательском режиме, и микроядро работает с ними, как с другими приложениями. Такие процессы иногда называются ***серверами.***

Такой подход позволяет разделить задачу разработки ОС на разработку ядра и разработку сервера. Это с одной стороны упрощает реализацию системы, обеспечивает её ёмкость, возможность настройки на требования конкретных приложений и среды, а также этот подход хорошо вписывается в концепцию распределения системы.

**Многопоточность**

Это технология, при которой процесс, выполняющий приложения, разделяется на несколько одновременно выполняющихся потоков.

***Поток***– это диспетчиризированная единица работы, включающая контекст процессора, а также свою собственную область стека.

Команды потока выполняется последовательно, и поток может быть прерван при переключении процессора на обработку другого потока. У потоков как правило общее адресное пространство, в отличии от процессов.

***Процесс*** – это набор из одного или нескольких потоков, а также связанных с этими потоками системными ресурсами, такими как область памяти для хранения кода и данных, открытые файлы, различные устройства.

Концепция процесса близка к концепции выполняющейся программы. Разбивая приложения на несколько потоков, программист получает преимущество модульности приложения и возможность управления связанными с приложением временными событиями.

Многопоточность оказывается полезной для приложений, выполняющих несколько независимых заданий, которые не требуют последовательного выполнения.

Потоки иногда называют ***облегченными процессами***.

**Симметричная многопроцессорность**

Этот термин относится к архитектуре аппаратного обеспечения компьютера, предполагающую наличие нескольких процессов (минимум 2), а также образу ОС, поддерживающей такую аппаратную архитектуру.

Симметричная многопроцессорность определяется следующими характеристиками:

1. В системе имеется несколько процессоров;
2. Процессоры соединены меду собой коммуникационной шиной и совместно используют одну и ту же основную память и одни и те же устройства ввода-вывода;
3. Все процессоры могут выполнять одни и те же функции.

Считается, сто многопроцессорные системы имеют несколько потенциальных преимуществ по сравнению с однопроцессорными:

- производительность;

- надежность;

- наращивание;

- масштабируемость.

В действительности эти преимущества не всегда реализуются. В частности, если задача строго линейна (т.е. не распараллеливается), то и выигрыша никакого не будет.

**Распределенные ОС**

Сущность этой концепции состоит в том, что создается так называемый кластер, состоящий из нескольких отдельных компьютеров, каждый из которых обладает собственной основной и вторичной памятью и своими устройствами ввода-вывода. Распределенные ОС создают видимость единого пространства основной и вторичной памяти и единой файловой системы.

К таким системам относятся системы типа клиент-сервер, но пока развитие таких систем находится в стадии развития.

**Объектно-ориентированный дизайн**

Помогает навести порядок в процессе добавления к основному небольшому ядру дополнительных модулей. Такой подход облегчает разработку пользовательских приложений, а также производить настройку ОС не нарушая её целостности. Классический пример таких систем –Windows.

**Концепция ОС на основе микроядра**

Первые ОС, которые разрабатывались в 50-х годах, можно отнести к классу *монолитных ОС.* В таких системах почти все процедуры могли вызывать одна другую. Такой подход практически не допускал расширяемости ОС.

Следующий этап, который во многом связан с машинами IBM, характерен разработкой *слоистых ОС*, основанных на иерархической организации функций. Причём взаимодействие возможно только с функциями, находящимися на соседних уровнях.

Аппаратное обеспечение

Пользователи

Файловая

Система

Обмен информацией между процессами

Управление вводом выводом и устройствами

Виртуальная память

Управление элементар-ными процессами

Пользовательский режим

Режим ядра

Рис.1.2. Структура слоистых ОС

Такой подход позволяет строить более надёжные ОС, позволяющие осуществлять модификацию, однако если потребуется ввести новые функции в слой, то необходимо вносить изменения и в соседние слои для возможности обращения к этим функциям. Другая проблема – это проблема безопасности, т. к. между слоями много точек обмена.

Основой концепции ОС на основе микроядра является то, что вертикальное расположение уровней заменяется горизонтальным, а приложения и услуги не являющиеся критическими, работают в пользовательском режиме. И хотя выбор того, что должно располагается в ядре, а что выносится за его пределы, зависит от архитектуры системы, общая тенденция такова, что многие службы, которые раньше размещались в ядре, теперь располагаются на уровне внешних подсистем, которые взаимодействуют с ядром и друг с другом, т. е. в ядре должны располагаться только самые важные функции. К таким подсистемам относятся драйвера устройств, файловые системы, менеджер виртуальной памяти, системы управления окнами, служба безопасности.

Аппаратное обеспечение

Микроядро

**…**



Обслужи-ваемый процесс

Режим ядра

Рис. 1.3. ОС на основе микроядра

Процессы выполняются на пользовательском уровне, взаимодействуя между собой как равноправные, и обычно взаимодействие осуществляется с помощью обмена сообщениями, которые передаются через микроядро. Микроядро выступает в роли посредника: оно подтверждает правильность сообщений, передаёт их от одного компонента другому, предоставляет доступ аппаратному обеспечению. Микро ядро выполняет и защитные функции: оно не пропускает сообщение, если такой обмен не разрешен.

Достоинства концепции микроядра.

1. *Единообразные интерфейсы.* Используются для запросов, генерируемых процессами. Процессам ненужно различать приложения, выполняемые на уровне ядра и на пользовательском уровне, так как доступ ко всем службам осуществляется только с помощью передачи сообщений.

2. *Расширяемость.* Имеется возможность добавлять в ОС новые услуги или сервисы, а также обеспечивать множественную реализацию сервисов в одной и той же функциональной области. Например, можно организовывать несколько различных способов хранения файлов на дисках. Добавление новой службы в ОС требует модификации лишь некоторых других служб и не требует изменять микроядро.

3. *Гибкость.* В ОC можно не только добавлять новые услуги, но и удалять некоторые из них. Это может потребоваться для получения компактной и эффективной версии. Но если пользователю не требуются некоторые подсистемы, занимающие большие объёмы памяти, он может скомпоновать компактную ОС по своим нуждам.

4. *Переносимость.* Программный код, который взаимодействует с аппаратными средствами, или большая его часть, находится в микроядре. Поэтому уменьшаются объёмы работ, связанных с переносом ОС на новую аппаратную платформу (процессор).

5. *Надёжность.* Чем больший код имеет программа, тем труднее её протестировать. Однако небольшое ядро ОС можно тщательно проверить. А небольшое число интерфейсов прикладного программирования позволяет реализовать подсистемы, работающие вне ядра, с достаточно качественным программным кодом. Имея стандартизованный набор интерфейсов, разработчик отдельного приложения не может повлиять на другие системные компоненты.

6*. Микроядро способствует поддержке распределенных ОС*. Сообщение, которое передается от обслуживаемых сервисов к обслуживающим должно содержать идентификатор запрашиваемой услуги. Если система такова, что все процессы и сервисы обладают в ней уникальными идентификаторами, то на уровне микроядра образуется единый образ системы. Процесс может отправлять сообщение, не зная, на какой именно машине выполняется приложение, к которому он обращается.

7. *Подход на основе микроядра хорошо функционирует среди объектно-ориентированных ОС.* Такой подход способствует более строгой разработке ядра и модульных расширений ОС. Перспективным считается подход, в котором сочетаются архитектура с микроядром, принципы объектно-ориентированных систем, которые реализуются с использованием компонентов. Компоненты – объекты с четко заданными интерфейсами, которые могут объединятся, образуя программы по принципу блоков.

Основным потенциальным недостатком микроядерявляется их низкая производительность. Создание сообщения и отправка его через микроядро с последующим получением и декодированием ответа занимает больше времени, чем непосредственно вызов сервиса.

Многое в обеспечении производительности зависит от функциональных возможностей ядра. Избирательное увеличение функциональности ядра приводит к снижению количества переключений между пользовательским режимом и режимом ядра.

Есть мнения и взгляды на то, чтобы сделать микроядро не больше, а наоборот – меньше. При этом повышается его гибкость и надёжность.

**Функции микроядра.**

1.***Низкоуровневое управление памятью.*** Для реализации в микроядре защиты на уровне процессов в нем должен обеспечиваться контроль над аппаратной организацией адресного пространства. Если микроядро будет отвечать лишь за отображение виртуальной страницы на физическую страницу, то блок управления памятью, включая систему защиты адресного пространства одного процесса от другого, а также алгоритм замены страниц и другие логические схемы страничной организации памяти можно реализовать вне ядра. Модуль виртуальной памяти принимает решение, когда загружать страницу в память, и какую из страниц, находящихся в памяти, следует заменить.

Приложения

Система страничной организации памяти

Микроядро

Возобновление работы

Вызов функции для работы с адресным пространством

Ошибка из-за отсутствия страницы

Рис. 1.4. Низкоуровневое управление памятью

Когда приложение обращается к странице, которая отсутствует в основной памяти, возникает прерывание из-за отсутствия страницы и управление перехватывается ядром. Ядро отправляет системе страничной организации памяти сообщение, в котором указывается запрашиваемая страница. Система страничной организации памяти может принять решение о загрузке данной страницы в оперативную память и выделения для этого физической страницы. Система страничной организации и ядро взаимодействуют между собой, чтобы логические операции, выполняемые системой страничной организации памяти, отображались в физическую память. Как только нужная страница станет доступна, то система страничной организации отправляет через микроядро сообщение приложению о том, что приложение может продолжить работу.

Считается, что в микроядре можно оставить только 3 операции для поддержки внешних систем страничной организации памяти и управления внешней памятью.

*Предоставление*. Процесс, который владеет адресным пространством, может предоставлять некоторые свои страницы другому процессу. Ядро удаляет эти страницы из адресного пространства первого процесса и передаёт их второму процессу.

*Отображение.* Процесс может отображать любые свои страницы в адресное пространство другого процесса. После чего оба процесса будут иметь доступ к этим страницам, то есть создаётся общая область памяти. Ядро не меняет информации о принадлежности страниц первому процессу, но выполняет отображение, предоставляя другому процессу доступ к этим страницам.

*Восстановление*. Процесс может восстановить любые страницы, предоставленные другим процессам или отображенные в их адресное пространство.

Вначале ядро определяет всю физическую память как единое адресное пространство, которым управляет основной системный процесс. При создании новых процессов страницы общего адресного пространства могут передаваться или отображаться в эти новые процессы. Такая схема позволяет одновременно поддерживать несколько схем организации виртуальной памяти.

2. ***Взаимодействие между процессами.*** Основной формой взаимодействия между процессами (потоками) являются сообщения. Сообщения включают в себя заголовок, в котором указаны идентификаторы процесса отправителя и процесса получателя, а также указатель на блок данных и некоторые управляющие сведения о процессе.

Взаимодействие между процессами основывается на относящихся к этим процессам *портам*. *Порт* – это очередь сообщений, предназначенная для определённого процесса. С портом связан список возможностей, в которых указано с какими процессами данный процесс может обмениваться информацией. Процесс может разрешить доступ к себе, отправив в ядро сообщение, в котором указана новая возможность порта.

Если адресное пространство в процессах не перекрывается, то передача сообщения от одного процесса другому – это копирование одной области памяти в другую.

3. ***Управление вводом-выводом и прерываниями.*** В микроядре имеется возможность обрабатывать аппаратные прерывания подобно сообщениям и включать в адресное пространство порта ввода-вывода, но не обрабатывает их. Микроядро распознает прерывания, но само не обрабатывает их. Оно генерирует сообщение процессу, работающему на пользовательском уровне и связанному с данным прерыванием, т. е. ядро преобразует прерывание в сообщение, но само в обработке аппаратно-зависимых прерываний не участвует.

В некоторых системах предлагается рассматривать аппаратное обеспечение как набор потоков, которые обладают своими идентификаторами и отправляют сообщение состоящее из идентификатора данного потока соответствующим потокам в пользовательских программах. Поток-получатель выясняет, является ли полученное сообщение прерыванием.

**Принципы построения ОС**

*1. Частотный принцип****.***

Этот принцип основан на выделении в алгоритмах программ и в обрабатываемых массивах действий и данных по частоте их использования. Действия и данные, используемые часто, располагаются в оперативной памяти, т.к. к ним необходим быстрый доступ. К тому же стремятся наиболее часто выполняемые операции оптимизировать по времени выполнения и по занимаемой памяти.

Следствия от применения частотного принципа – это применение многоуровневого планирования при организации работы ОС. На уровень долгосрочного планирования выносятся редкие и длинные операции управления деятельностью системы. Краткосрочному планированию подвергаются часто используемые и короткие операции.

*2.Принцип модульности****.***

Это принцип в равной степени отражает технологические и эксплуатационные свойства. Наибольший эффект от его использования достигается в том случае, когда принцип одновременно распространен на ОС, аппаратуру и прикладные программы.

Под модулем в общем случае понимают функциональный элемент рассматриваемой системы, имеющей законченное оформление и выполненный в пределах требования ОС, а также средства сопряжения с подобными элементами или элементами более высокого уровня данной или другой системы.

Модуль предполагает лёгкий способ его замены на другой модуль, при наличии заданных интерфейсов. Чаще всего разделение на модули происходит по функциональному признаку.

Модули могут быть восстанавливаемыми и невосстанавливаемыми. Если модуль после окончания работы не восстанавливается в исходное состояние, то он называется однократным. Если модуль в процессе работы искажает своё состояние, но перед окончанием работы восстанавливается в исходное состояние, то его называют многократным. Особое значение при построении ОС имеют модули, называемые параллельно используемыми или реентерабельными. Каждый такой модуль может использоваться одновременно несколькими программами. Это позволяет хранить в памяти только одну копию такого модуля.

*3.Принцип функциональной избирательности.*

В ОС выделяется некоторая часть особо важных модулей, которые должны быть в оперативной памяти постоянно для эффективной организации вычислительного процесса. Эту часть обычно называют ядром ОС.

При формировании ядра необходимо удовлетворить двум противоречивым требованиям:

*–* в состав ядра должны войти наиболее часто используемые модули;

*–* количество модулей должно быть таким, чтобы не занимать много оперативной памяти;

В состав ядра входят модули по управлению системой прерываний, средства перевода процессов с одного состояния в другое, средства распределения оперативной памяти. Программы, входящие в состав ядра, обычно загружаются в оперативную память и называются резидентными.

Помимо резидентных программ существуют транзитные модули или программы, которые загружаются только при необходимости и могут перекрывать в оперативной памяти друг друга.

*4.Принцип генерируемости.*

Определяет такой способ исходного представления ОС, который позволял бы настраивать эту системную программу, исходя из конкретной конфигурации используемой машины и круга решаемых задач.

Процедура генерации проводится достаточно редко, а процесс генерации осуществляется с помощью специальной программы-генератора и входного языка для неё, позволяющего описывать программные возможности системы и конфигурацию машины. В результате генерации получается полная версия ОС. Исходный набор программ ОС, из которого производится генерация называется дистрибутивом.

*5.Принцип функциональной избыточности.*

Предполагает возможность проведения одной и той же работы различными средствами, т.е. ОС допускает альтернативное выполнение одних и тех же заданий в различных режимах своего функционирования.

*6.Принцип по умолчанию.*

Принцип основан на хранении в системе некоторых базовых описании, структур процесса, модулей, конфигурации оборудования и данных, определяющих прогнозируемые объемы требуемой памяти, времени счета программы, потребности во внешних устройствах, которые характеризуют пользовательские программы и условия их выполнения.

Эту информацию пользовательская система использует в качестве заданной, если она не будет определена или сознательно конкретизирована. В целом применение этого принципа позволяет сократить число параметров, устанавливаемых пользователем, когда он работает с системой.

*7.Принцип перемещаемости.*

Предусматривает построение модулей таким образом, что их исполнение не зависит от места расположения в оперативной памяти. Настройка текста модуля в соответствии с его расположением в памяти осуществляется специальными механизмами либо непосредственно пред выполнением программы, либо по мере по мере ее выполнения.

Настройка заключается в определении фактических адресов, используемых в адресных частях команды, и определяется применяемым в конкретной машине способом адресации и алгоритмом распределения оперативной памяти, принятым для данной ОС.

Этот принцип целесообразно распространять и на пользовательские программы.

*8.Принцип защиты.*

Определяет необходимость разработки мер, ограждающих программы и данные пользователя от искажения и нежелательного влияния друг на друга, а также пользователя на ОС и наоборот. Особенно трудно обеспечить защиту, когда используется разделение ресурсов.

Программы должны быть гарантированно защищены как при выполнении, так и при хранении, хотя попыток испортить и нанести нежелательный эффект пользовательским программам совершается множество.

Реализуется несколько подходов для обеспечения защиты.

Одним из направлений является реализация двухконтекстности работы процессора: в каждый момент времени процессор может выполнять программу из состава ОС либо прикладную или служебную программу, не входящую в состав ОС.

Второе направление состоит в том, чтобы гарантировать невозможность непосредственного доступа к любому разделяемому ресурсу со стороны пользовательских и служебных программ, для чего в состав машинных команд вводятся специальные привилегированные команды, управляющие распределением и использованием ресурсов. Такие команды разрешается выполнять только ОС. Контроль за выполнением привилегированных команд производится аппаратно.

Для реализации принципов защиты может использовать контекстный механизм защиты данных и текста программ, находящийся в операционной памяти. Для программ пользователей выделяются определенные участки памяти и выход за пределы этих участков приводит к возникновению прерываний по защите. Механизм контроля реализуется аппаратным способом, путем применения ограничительных регистров или ключей памяти.

Третье направление реализует механизм защиты данных, хранящихся в памяти, используются подходы, основанные на разграничении прав доступа, введении паролей, контроле за правильной интерпретацией данных, записанных в файле.

*9.Принцип независимости программ от внешних устройств.*

Позволяет выполнять операции управления внешними устройствами независимо от их конкретных физических характеристик. Связь программ с конкретными устройствами производится не на уровне трансляции программ, а в период планирования ее выполнения. При подключении новых устройств или замене существующих, текст программ не изменяется, а осуществляется подключение нового устройства к ОС путём подключения специальной программы, обеспечивающей взаимодействие ОС с внешним устройством. Такие программы называются драйверами.

*10.Принцип открытой и наращиваемой операционной системы.*

Открытая ОС доступна пользователю и специалисту, обслуживающему машину, а возможность наращивания или модификации операционной системы позволяет использовать не только возможности генерации, но вводить в операционную систему новые модули или модифицировать существующие.

***Тема 6.Основы операционных систем***

**Понятие процесса**

В предметной области системного программирования понятие процесса является базовым, но вместе с тем недостаточно формально определено. Существуют определения процесса как формального, так и неформального свойства. Понятие процесса является определенным видом абстракции, которую по-разному используют и трактуют разные группы лиц.

Многие архитектуры современных вычислительных машин являются многопроцессорными. Процессор - устройство в составе ЭВМ, способное автоматически выполнять допустимые для него действия в некотором определенном порядке, т.е. по определенной программе, хранящейся в памяти, и непосредственно доступной устройству. Между процессорами в системе существуют информационные и управляющие связи.

Каждый процессор - объект в системе, которым хотели бы воспользоваться одновременно несколько пользователей для исполнения своей программы. В отношении каждого пользователя, претендующего на исполнение программы на некотором процессоре, и системы, распределяющей этот процессор среди многих пользователей, вводится понятие “процесс”.

В общем случае процесс - некоторая деятельность, связанная с исполнением программы на процессоре.

Согласно стандартизованного определения процесс - система действий, реализующая определенную функцию в вычислительной системе и оформленная так, что управляющая программа вычислительной системы может перераспределять ресурсы этой системы в целях обеспечения мультипрограммирования.

При выполнении программы могут потребоваться результаты других процессов или процессоров, или другие ресурсы. Следовательно, ходом развития процесса нужно управлять.

Управление процессами, как и в отношении каждого, так и в отношении их совокупности - функция ОС.

При выполнении программ на центральном процессоре чаще всего различают следующие характерные состояния.

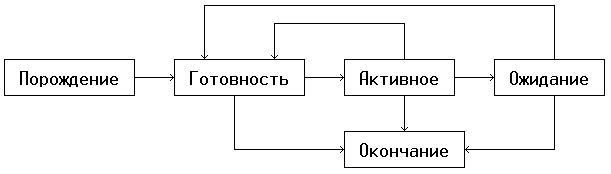


Рис. 2.1. Состояния процесса и переходы

Порождение - подготовка условий для первого исполнения на процессоре.

Активное состояние, или состояние "счет" - программа выполняется на процессоре.

Ожидание - программа не исполняется на процессоре по причине занятости какого-либо требуемого ресурса.

Готовность - программа не исполняется, но для использования предоставлены все необходимые в текущей момент ресурсы, кроме ЦП.

Окончание - нормальное или аварийное окончание исполнения программы, после которого процессор и другие ресурсы ей не предоставляются.

Процесс находится в каждом или некоторых из своих допустимых состояний в течение некоторого времени, после чего переходит в другое допустимое состояние.

Процессы определяются рядом временных характеристик. В некоторый момент времени процесс может быть порожден, а через некоторое время закончен. Интервал между этими моментами называют интервалом существования процесса. Длительность интервала существования процесса, в общем случае, непредсказуема.

Отдельные виды процессов требуют такого планирования, чтобы гарантировать окончание процесса до наступления некоторого конкретного времени. Это процессы реального времени.

В другой класс входят процессы, время существования которых должно быть не более интервала времени допустимой реакции ЭВМ на запросы пользователя. Это интерактивные процессы. Интервал времени является допустимым, если он не раздражает пользователя.

Остальные процессы относятся к классу пакетных.

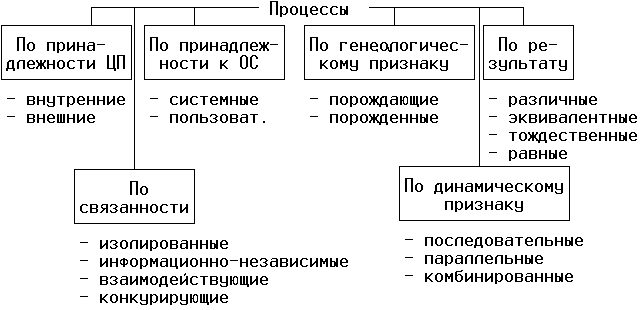


Рис. 2.2. Классификация процессов

Сравнение процессов может быть произведено с использованием понятия "трасса" - порядок и длительность пребывания процесса в допустимых состояниях на интервале существования.

Два процесса, имеющие одинаковый конечный результат обработки одних и тех же исходных данных по одной той же или даже различным программам, называют эквивалентными. Трассы эквивалентных процессов в общем случае не совпадают. Если в каждом из эквивалентных процессов обработка данных происходит по одной и той же программе, но трассы при этом в общем случае не совпадают, то такие процессы называют тождественными. При совпадении трасс у тождественных процессов их называют равными. Во всех остальных случаях процессы всегда различны.

Проблематичность управления процессами заключается в том, что в момент порождения процессов их трассы неизвестны. Также требуется учитывать, как соотносятся во времени интервалы существования процессов.

Если интервалы существования двух процессов не пересекаются во времени, то такие два процесса называют последовательными друг относительно друга. Если на рассматриваемом интервале времени существуют одновременно два или более процесса, то они на этом интервале являются параллельными. Если на рассматриваемом интервале находится хотя бы одна точка, в которой существует один процесс, но не существует другой, и есть хотя бы одна точка, в которой оба процесса существуют одновременно, то такие два процесса называют комбинированными.

Процессы называются взаимосвязанными, если между ними с помощью системы управления процессами поддерживается какого-либо рода связь: функциональная, пространственно-временная, управляющая, информационная и т.д. В противном случае процессы являются изолированными. При наличии между процессами управляющей связи устанавливается соотношение вида “порождающий-порождённый”.

Если взаимосвязанные процессы при развитии используют совместно некоторые ресурсы, но не связаны между собой информационно, то такие процессы называют информационно-независимыми, а при наличии информационной связи процессы называются взаимодействующими. Схемы механизмов установки таких связей различны, они обусловлены динамикой процессов и выбранным способом связи: явным (явный обмен сообщениями между процессами), или неявным (с помощью разделяемых структур данных). Когда необходимо подчеркнуть связь между взаимосвязанными процессами по ресурсам, их называют конкурирующими.

Управление взаимосвязанными процессами основано на контроле и удовлетворении определенных ограничений, которые накладываются на порядок выполнения таких процессов. Данные ограничения определяют виды отношений, допустимых между процессами, и составляют в совокупности синхронизирующие правила.

Отношение предшествования для двух процессов означает, что первый процесс должен переходить в активное состояние раньше второго.

Отношение приоритетности означает, что некоторый процесс с определённым приоритетом может быть переведён в активное состояние, если в состоянии готовности нет процессов с большим приоритетом, а процессор либо свободен, либо используется программой с меньшим приоритетом.

Отношение взаимного исключения. Процессы используют общий ресурс. Совокупность действий над этим ресурсом одного процесса называют критической областью. Критическая область одного процесса не должна выполняться одновременно с критической областью другого процесса для этого же ресурса.

Проблема в реализации синхронизирующих правил в составе системы управления процессами обусловлена динамикой процессов, неопределенностью и непредсказуемостью порядка и частотой перехода процессов из состояния в состояние по мере их развития.

В отношении каждой совокупности взаимодействующих процессов приходится решать задачу синхронизации, которая требует определённого порядка выполнения процессов с целью установления требуемого взаимодействия.

**Понятие ресурса**

Ресурс - всякий потребляемый объект (независимо от формы его существования), обладающий некоторой практической ценностью для потребителя.

Ресурсы различаются по запасу выделяемых единиц ресурса и бывают в этом смысле исчерпаемые и неисчерпаемые. Исчерпаемость ресурса, как правило, приводит к конфликтам в среде потребителей. Для регулирования конфликтов ресурсы должны распределяться между потребителями по каким-то правилам, в наибольшей степени их удовлетворяющим.

Вычислительную систему или ЭВМ можно представить как ограниченную последовательность функциональных элементов, обладающих потенциальными возможностями выполнения с их помощью или над ними действий, связанных с обработкой, хранением или передачей данных.

Такие элементы потребляются другими элементами, являющимися в общем случае пользователями.

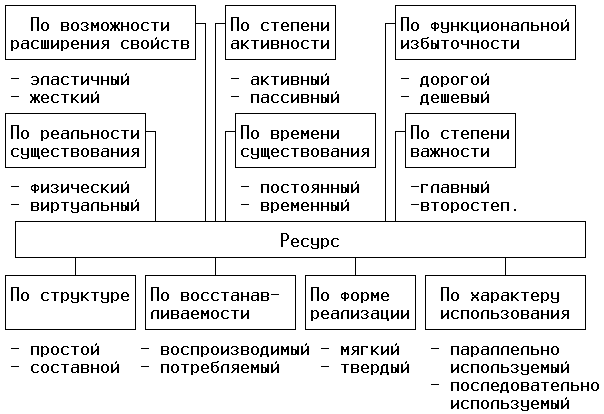


Рис. 2.3. Классификация ресурсов

Уровень детализации элементов, выделяемых по запросам для использования в системе, может быть различным. Считается, что потребителем выделяемых элементов вычислительной системы будут являться процессы, развивающиеся в ней. Все выделяемые по запросам от процессов элементы системы отождествляются с понятием ресурсов.

В соответствие с ГОСТ ресурсом является средство вычислительной системы, которое может быть выделено процессу на определенный интервал времени.

Под физическим понимают ресурс, который реально существует и при распределении его между пользователями обладает всеми присущими ему физическими характеристиками.

Виртуальный ресурс - это некоторая модель физического ресурса. Виртуальный ресурс не существует в том виде, в котором он проявляет себя пользователю. Как модель виртуальный ресурс реализуется в некоторой программно-аппаратной форме. В этом смысле виртуальный ресурс существует. Однако виртуальный ресурс может предоставить пользователю не только часть тех свойств, которые присущи объекту моделирования, т.е. физическому ресурсу, но и свойства, которые ему не присущи. Однако пользователь будет воспринимать их, как реально существующие.

Признак возможности расширения свойств характеризует ресурс с точки зрения возможности построения на его основе некоторого виртуального ресурса. Физический ресурс, который допускает виртуализацию, называется эластичным. Жестким называется физический ресурс, который по своим внутренним свойствам не допускает виртуализации.

При использовании активного ресурса он способен выполнять действия по отношению к другим ресурсам (или даже в отношении самого себя) или процессам, которые в общем случае приводят к изменению ресурса. (Например, ЦП - активный ресурс).

Пассивный ресурс не обладает таким свойством. (Например, память).

Разделение ресурсов на дорогие и дешевые связано с реализацией принципа функциональной избыточности при распределении ресурсов. Перед пользователем стоит задача выбора - получить быстро требуемый ресурс и дорого заплатить за такую услугу, либо подождать выделения требуемого ресурса и после его использования заплатить меньше. При наличии в системе альтернативных ресурсов вводятся и различные цены за их использование.

В отношении каждого ресурса процесс-пользователь выполняет три типа действий: запрос, использование, освобождение.

Если при распределении системой ресурса допускается многократное выполнение действий в последовательности запрос-использование-освобождение, то такой ресурс называют воспроизводимым. Если считать, что при этом ресурс не утрачивает своих свойств, то время его существования равно бесконечности.

В отношении некоторых категорий ресурсов порядок действий таков: освобождение-запрос-использование, после чего ресурс, который в данном случае называют потребляемым, изымается из сферы потребления.

Правила распределения ресурсов обуславливаются параллельной или последовательной схемой использования распределяемого между несколькими процессами ресурса. Последовательная схема предполагает, что в отношение некоторого ресурса, который называют последовательны, допустима строго последовательная во времени последовательность действий: запрос, исполнение, освобождение – каждым процессом-потребителем этого ресурса. Для параллельных процессов такие цепочки действий называются критическими областями и при выполнении должны удовлетворять правилу взаимного исключения.

Последовательно выполняемый ресурс, разделяемый несколькими параллельными процессами, поэтому называется критическим ресурсом. Параллельная же схема предполагает одновременное использование одного и того же ресурса несколькими процессами. Но такое использование не должно вносить каких либо изменений в логику развития каждого из процессов.

По форме реализации различают твердые и мягкие ресурсы. Под твердыми понимаются аппаратные компоненты машины, а также человеческие ресурсы, остальные ресурсы относятся к мягким. Различным по отношению к твердым и мягким ресурсам является подверженность к сбойным или отказным ситуациям, а также последующее восстановление работоспособности.

В классе мягких ресурсов выделено два типа:

- программные;

- информационные.

Если мягкий ресурс допускает копирование и эффект от использования ресурса-оригинала и ресурса-копии идентичен, то такой ресурс называется программным, в обратном случае – информационным. Мягкие информационные ресурсы либо принципиально не допускают копирования, либо допускают копирование, но оно является функцией времени (сообщения, сигналы, запросы на прерывание и т.д.).

**Концепция виртуализации**

Концепция виртуализации рассматривается в ОС как средство уменьшения конфликтов при взаимодействии процессов и распределении ресурсов, облегчая работу пользователя с вычислительной системой при его обращении к тем или иным ресурсам.

Виртуализация достигается на основе централизованной схемы распределения ресурсов. При такой организации работы обеспечиваются *две формы виртуализации*:

1. Пользователь обеспечивается при обслуживании своего процесса ресурсом, который реально не существует, или существует, но с ухудшенными характеристиками.

2. Для нескольких параллельных процессов создается иллюзия одновременного использования того, что одновременно в реальной системе существовать не может.

Целью виртуализации является предоставление пользователям ресурсов с характеристиками, в наибольшей степени их удовлетворяющими и позволяющими снять ограничения на количество распределяемых ресурсов, что позволяет увеличить скорость развития процесса, организовать более гибкое управление процессами и снять ограничение на их количество.

Виртуализация обеспечивается аппартно-программными средствами.

*Примеры виртуализации и виртуальных ресурсов:*

1. Построение на основе одного интервального таймера произвольного числа таймеров, которые предоставляются пользователям в монотонное использование.

2. Применение мультиплексных каналов при работе с внешними устройствами. Имеется одно устройство, организующее связь пользовательских программ с внешними устройствами. Для каждого устройства каналом выделяется квант времени. Величина кванта времени для различных устройств может быть разной. Устройствам при таком режиме создаётся видимость, что они непрерывно обмениваются информацией с пользовательскими режимами.

3. Организация на базе одного из устройств, имеющего определенный способ доступа, имитацию работы других устройств с другими способами доступа.

4. Использование так называемых нулевых устройств. Такие устройства реально не существуют, но создаётся видимость их существования. При передаче данных на такое устройство они поглощаются, а при чтении данных с такого устройства сразу выдается признак конца файла.

5. Понятие спулинга при работе с внешними устройствами. При этом каждое внешнее устройство моделируется некоторой областью внешней памяти. Каждая такая область представляет собой буфер между процессом и реальным устройством. Если этот буфер выполняет роль выходного, то в нем накапливаются данные по мере выполнения процессором команд вывода. При этом не производится вывода, но у процесса появляется ощущение, что он что-то вывел. Реальный вывод производится под контролем системы. Входной буфер работает таким же образом. В нем накапливаются данные, вводимые от внешних устройств, а затем эти данные передаются процессу. Типичный пример устройства, в котором применяется спулинг – это устройство печати.

Рассмотренные примеры иллюстрируют возможность построения и использования виртуальных ресурсов. Каждый виртуальный ресурс подобного рода генерируется некоторым процессом. Управление этим процессом осуществляется со стороны ОС, которая в этом смысле выполняет роль распределителя реального ресурса, на базе которого строится один или несколько виртуальных. Если поддерживается несколько виртуальных ресурсов, то в составе распределения реализуют дисциплины распределения таких ресурсов среди процессов-пользователей.

Одним из наиболее типичных или законченных примеров концепции виртуализации является понятие *виртуальной машины*. Любая ОС, являясь средством распределения ресурсов, организует по определённым правилам управление процессами на базе скрытой аппаратной части, создавая у пользователя видимость виртуальной машины.

Типичный пользователь видит и использует виртуальную машину как некоторое устройство, способное воспринимать его программы, написанные на определённом языке программирования, выполнять их и выдавать результаты. При таком подходе пользователя не интересует структура машины и способы эффективного использования её составных частей. Пользователь работает с машиной в терминах, применяемых в языке программирования.

Зачастую виртуальная машина, предоставленная пользователю, воспроизводит реальную архитектуру машины, но элементы архитектуры в таком представлении выступают с новыми или улучшенными характеристиками.

Идеальная в представлении пользователя архитектура виртуальной вычислительной машины имеет следующий состав:

– бесконечная по объему память с произвольно выбираемым наиболее удобным для пользователя доступом к объектам, хранимым в памяти

– один или несколько процессоров, способных выполнять действия, выражаемые пользователем в терминах некоторых удобных для него языков программирования

– произвольное количество внешних устройств с удобным способом доступа и предоставление информации, передаваемой через эти устройства или хранимой ими без каких-либо существенных ограничений на объем информации

Концепция виртуализации нашла широкое применение при проектировании и реализации ОС. Наиболее рационально представить структуру системы в виде определенного набора планировщиков процессов и распределителей ресурсов, которые часто называются ***мониторами.***

*Монитор–* распределитель некоторого ресурса, который может на основании некоторой организации работы обеспечить ту или иную степень виртуализации при распределении эластичного ресурса.

Использование концепции виртуализации положено в основу восходящего метода проектирования и разработки ОС. ОС строится как иерархия вложенных друг в друга виртуальных машин. Низшим уровнем иерархии являются аппаратные средства машины. Следующий уровень *–* программный. Он совместно с нижним уровнем обеспечивает достижение машиной новых свойств (виртуальная машина I уровня). Относительно этой виртуальной машины разрабатывается новый программный слой, и получается виртуальная машина II уровня. Последовательно наращивая уровни, можно реализовать виртуальную машину с требуемыми свойствами. При этом каждый новый слой позволяет расширить функциональные возможности по обработке данных и позволяет организовать достаточно простой доступ к нижним уровням. Применение метода иерархического построения виртуальных машин позволяет систематизировать проект, повысить надежность сложной программной системы, уменьшить сроки разработки. Однако при этом имеется ряд проблем, основная из них: определение свойств и количества уровней виртуализации и определение правил внесения на каждый уровень необходимых частей ОС.

Имеются ряд правил по формированию уровней и способов их взаимодействия.

1. На каждом уровне ничего не известно о свойствах и о существовании более высоких уровней.

2. На каждом уровне ничего не известно о внутреннем строении других уровней. Связь между ними осуществляется только через жесткие, заранее определенные сопряжения.

3. Каждый уровень представляет собой группу модулей, некоторые из которых являются внутренними и не доступны для других уровней. Имена остальных модулей известны на следующем, более высоком уровне, и представляют собой сопряжение с этим уровнем.

4. Каждый уровень располагает определенными ресурсами и либо скрывает их от других уровней, либо представляет другим уровням их абстракции (виртуальные ресурсы).

5. Каждый уровень может обеспечивать некоторую абстракцию данных в системе.

6. Предположения, которые делаются на каждом уровне относительно других уровней, должны быть минимальными.

7. Связь между уровнями ограничена явными аргументами, передаваемыми с одного уровня на другой.

8. Недопустимо совместное использование несколькими уровнями глобальных данных.

9. Каждый уровень должен иметь высокую прочность и слабое сцепление с другими уровнями.

10. Всякая функция, выполняемая уровнем абстракции, должна иметь единственный вход.

**Дисциплины распределения ресурсов**

Идея мультипрограммирования связана с наличием очередей процессов. Процессор в многозадачных системах по очереди предоставляется процессам.

Использование многими процессами одного ресурса, который в данный момент времени может обслуживать только один процесс, осуществляется с помощью дисциплин распределения ресурсов. Их основой является:

1. *Дисциплины формирования очередей на ресурс*⎯ это совокупность правил, определяющих размещение процессов в очереди.
2. *Дисциплина обслуживания очереди*⎯ совокупность правил извлечения одного из процессов из очереди с последующим предоставлением ему ресурса для использования.

Основным конструктивным и согласующим элементом при реализации дисциплин диспетчеризации является очередь.

Дисциплины формирования очередей разделяют на два класса:

– *статический*: приоритеты назначаются до выполнения задания

– *динамический:* приоритеты назначаются в процессе выполнения задания

На практике чаще всего встречаются следующие *дисциплины распределения или обслуживания ресурсов:*

1. Обслуживание в порядке поступления. Все заявки поступают в конец очереди. Первыми обслуживаются заявки из начала очереди.

обслуживание

очередь

запрос

1. Обслуживание в порядке, обратном порядку поступления. Данная дисциплина является основой построения стековой памяти.

обслуживание

очередь

запрос

Общим для 1 и 2 является относительная простота и определённая справедливость в обслуживании всех потоков, поступающих в систему. Среднее время ожидания запросов в очереди при некотором установившемся темпе обслуживания и поступления является одинаковым, независимо от характеристик процессов пользователей. Помимо этого, FIFO обеспечивает минимизацию дисперсии времени ожидания.

1. Круговой циклической алгоритм.

квант

обслуживания tk

Новый

запрос

Обслуженный

запрос

ресурс

очередь

1. В основу этой дисциплины обслуживания положена дисциплина FIFO, но время обслуживания каждого процесса здесь ограничено и определяется квантом времени обслуживания. Если запрос был обслужен полностью в течение этого кванта, он покидает очередь, иначе он помещается в конец очереди. А если не успел, то поступает опять в конец очереди. Эта дисциплина широко применяется в системах распределения времени. Хотя в этой дисциплине нет явных приоритетов, автоматически получают преимущество процессы, не требующие долгого обслуживания. Короткие процессы имеют меньшее среднее время пребывания в очереди по сравнению с длинными. Чем меньше квант времени, тем выше благоприятствование коротким процессам, но уменьшение кванта времени ведёт к увеличению затрат, связанных с переключением с процесса на процесс, что особенно неблагоприятно для длинных процессов. Это одноочередная дисциплина*.*

В ОС также широко используются многоочередные дисциплины.

**Концепция прерывания**

Реализация многопрограммного режима работы ЭВМ основана на использовании прерываний. Так, программа, обслуживаемая процессором, прерывается из-за отсутствия данных, подлежащих обработке в оперативной памяти. Программа, обслуживаемая процессором, может быть прервана более приоритетной программой. Из всего многообразия различных причин прерывания необходимо выделить два вида системных причин прерывания: первого и второго рода.

*Системные причины прерывания первого рода* возникают в том случае, когда у процесса, находящегося в активном состоянии, возникает потребность получить некоторый ресурс или отказаться от него либо выполнить над ресурсом какие-либо действия. Эти причины появляются и тогда, когда процесс порождает, уничтожает и выполняет какие-либо действия в отношении других процессов. При таких прерываниях возникает необходимость в явной форме выражать требование на прерывание процессом самого себя. Необходимо установление связи типа «вызывающий — вызываемый». Установка такой связи реализуется, как правило, в форме макрокоманд, представленных в пользовательской программе. При выполнении таких макрокоманд происходит переключение процессора с обработки программы пакета на работу операционной системы, которая подготавливает и обеспечивает выполнение соответствующего прерывания. К этой группе относятся и так называемые внутренние прерывания, которые связаны с работой процессора и являются синхронными с его операциями. К таким прерываниям относится арифметическое переполнение, исчезновение порядка в операциях с плавающей запятой, обращение к заишщеннрму массиву оперативной памяти и др.

*Системная причина прерывания второго рода* обусловлена необходимостью проведения синхронизации между параллельными процессами. Процессы, порожденные и подчиненные операционной системе, по мере их окончания или при какой-то другой ситуации вырабатывают сигнал прерывания. Например, по мере окончания внешнего процесса пересылки массива данных с МД в оперативную память вырабатывается сигнал прерывания. Этот сигнал прерывает исполнение обслуживаемой программы процессором, которое происходит «без ее ведома», т. е. асинхронно.

При обработке каждого прерывания необходимо выполнять такую последовательность действий:

1) восприятие запроса на прерывание;

2) запоминание состояния прерванного процесса, определяемое прежде всего значением счетчика команд. Оно должно отражать и признак команды, после выполнения которой произошло прерывание, содержимое регистров общего назначения, режим работы процессора (с позиции обслуживания процессором пользовательской или системной программы в момент прерывания);

3) передача управления прерывающей программе, для чего счетчик команд СК должен быть занесен адрес, который, как правило, является уникальным для каждого типа прерывания;

4) обработка прерывания;

5) восстановление нормальной работы.

В большинстве ЭВМ этапы 1—3 реализуются аппаратными средствами, а этапы 4 и 5 — операционной системой, ее блоком программ обработки прерываний.

Наряду с описанными функциями, реализуемыми на аппаратном уровне, обработка, прерываний реализуется в ОС на уровне системных программ. Такие функции представлены в прерывающих программах. Их количество определяется количеством уровней прерывания.

Например, обработчик внешних прерываний, обработчик программных прерываний. Прерывающая программа, обслуживающая запросы по уровню ввода-вывода называется супервизором ввода-вывода. Прерывающие программы по уровню ОС называются программами супервизора.

Каждая прерывающая программа соответствует определенному уровню прерываний. Последний объединяет ряд запросов прерываний, обслуживаемых одной прерывающей программой. При таком решении кроме выбора прерывающей программы необходимо выбирать в уровне наиболее приоритетные запросы, требующие обслуживания. Эта функция часто реализуется в системных прерывающих программах.

После окончания обработки прерываний происходит восстановление нормальной работы ЭВМ. Это восстановление происходит передачей управления системной программе, выполняющей функции диспетчера. Последняя получает управление в тех случаях, когда результатом обработки прерывания является вывод некоторой задачи из состояния ожидания в состояние готовности или постановкой текущей задачи в состояние ожидания. Программа «Диспетчер» анализирует состояние задач в системе и передает управление готовой к выполнению задаче с наивысшим приоритетом. Если задачи в состоянии готовности отсутствуют, то центральный процессор переводится в состояние ожидания до возникновения очередного прерывания.

***Тема 7. Процессы***

**Состояние процессов**

Процесс – это система действий, реализующая определенную функцию в вычислительной системе и оформленная так, что управляющая программа вычислительной системы может перераспределять ресурсы этой системы в целях обеспечения мультипрограммирования.

Для процесса при выполнении на центральном процессоре можно выделить следующие характерные отдельные состояния.

**Новый процесс**

**Готовый к выполнению**

**Выполняю-щийся**

**Завершенный**

**Блокированный**

Освобождение

Диспетчеризация

Ожидание события

Событие

TimeOut

Ввод в систему

Рис. 3.1. Схема переходов процесса из состояния в состояние

*Новый*. Только что созданный процесс, который еще не помещен ОС во множество процессов.

*Выполняющийся.* В данный момент работающий процесс.

*Готовый к выполнению.* Процесс будет запущен, как только представится возможность.

*Блокированный.* Процесс, который ожидает некоторого события.

*Завершающийся.* Процесс, удаленный из множества запущенных процессов.

Запуск процесса состоит из 2-х этапов.

1. Присвоение процессу идентификатора и формирование всех таблиц, необходимых процессу.

2. Распределение памяти под процесс.

Рассмотрим систему переходов из состояние в состояние.

*Нулевое состояние- «состояние новый».* Этот переход происходит при наступлении одного из следующих событий.

1. Если ОС работает в пакетном режиме и для обработки поступает управляющий поток пакетных заданий.

2. Если система работает в интерактивном режиме и в систему с терминала входит новый пользователь.

3. Создание ОС процесса, необходимого для выполнения служебных функций.

4. Порождение одного процесса другим.

*Переход из «новый» в «готовый».* ОС осуществляет переход, когда будет готова к обработке дополнительных процессов. В большинстве систем существует ограничения на количество выполняющихся процессов или на объем виртуальной памяти.

*Переход из «готовый» в «выполняющийся».* Происходит, когда ОС выбирает новый процесс для запуска. Выбирается один из процессов, находящихся в состоянии «готовый», в соответствии с какой-то дисциплиной в обслуживании.

*Переход из «выполняющийся» в «готовый».* Чаще всего происходит, когда процесс отрабатывает максимальный промежуток времени, отведенный для непрерывной работы одного процесса.

*Переход из «выполняющийся» в «блокированный».* Процесс переводится в заблокированное состояние, если для продолжения его работы требуется какое-либо событие. Например, процесс может запросить какой-либо ресурс, который временно недоступен или требуется выполнить какое-либо действие, необходимое для продолжения работы процесса, например, операцию ввода-вывода.

*Переход «заблокированный»-*«*готовый»*. Осуществляется, когда происходит ожидаемое событие.

*Переход из «выполняющийся» в «завершающийся».* Выполняется тогда, когда процесс сигнализирует об окончании своей работы, или ОС прекращает его выполнение в силу каких-то причин.

***Основные причины создания процессов следующие.***

1. Новое пакетное задание. Готовясь принять на обработку новое задание. ОС считывает очередные команды управления заданием.

2. Вход в систему в интерактивном режиме, когда в систему с терминала подключается новый пользователь.

3. Создание ОС процесса нужной для работы некоторой служебной ОС может создать процесс, для выполнения некоторой операции.

4. Порождение одного процесса другим с целью структуризации программы или использования возможности параллельных вычислений, программа может создавать другие процессы.

***Основные причины завершения процессов следующие.***

1. Обычное завершение.

2. Превышение программой времени, отведенной на её выполнение.

3. Недостаточный объем памяти.

4. Выход за пределы отведенной области памяти.

5. Ошибки защиты.

6. Арифметические ошибки (деление на ноль; использование чисел, выходящих за разрядную сетку).

7. Излишнее ожидание (т.е. процесс ждет наступление события больше, чем указано в параметрах системы).

8. Ошибки ввода-вывода.

9. Неверная команда.

10. Команда с недоступными привилегиями.

11. Неправильное использование данных.

12. Вмешательство оператора или ОС.

13. По требованию родительского процесса.

ОС может завершить любой процесс, находящийся в любом состоянии по своему усмотрению.

**Описание процессов**

ОС можно представить как некий механизм, управляющий тем, как процессы используют системные ресурсы. И т.к. одна из задач ОС- управление процессами и ресурсами, то ОС должна располагать информацию о текущем состоянии каждого ресурса и процесса. Для этих целей ОС создает и поддерживает таблицы с информацией по каждому объекту управления. Общая структура таких таблиц может иметь вид, представленный на рис. 3.2.

**Структуры управления процессами**

Для управления ОС должна знать где находится процесс и атрибуты нужные для управления процессами. Набор атрибутов, которые используется ОС называется управляющим блоком процесса. Имеется понятие – образ процесса. В образ входит следующая информация.

1. Данные пользователя – это допускающая изменения часть пользовательского пространства, а также данные программы, пользовательский стек и модифицированный код

2. Пользовательская программа – программный код, который надо выполнить

3. Системный стек. С каждым процессом связаны один или несколько системных стеков. Стек используется для хранения параметров, адресов вызовов процедур и системных функций

4. Управляющий блок процесса – данные нужные ОС для управления процессом

Местонахождение образа процесса зависит от системы управления памятью. Но для управления процессом надо, чтобы какая-то его часть находилась в основной памяти. А чтобы запустить процесс, его надо полностью загрузить в основную и виртуальную память. Образы состоят из блоков, которые необязательно должны располагаться последовательно.

Атрибуты процессов.

1. Идентификаторы процесса:

- идентификатор данного процесса;

- идентификатор родительского процесса;

- идентификатор пользователя.

2. Информация о состоянии процесса:

- регистры, доступные пользователю (те, к которым можно обращаться с помощью машинных команд);

- управляемые регистры и регистры состояния (те, которые управляют работой процесса). Должны быть счетчик команд, который содержит адрес очередной команды, коды условия, которые отражают результат выполнения арифметической или логической команды и информация о состоянии;

- указатели на стек – указывает на вершину стека, с процессом связывается один или несколько стеков.

3. Управляющая информация процесса:

а) информация по планированию и состоянию: состояние процесса, приоритет, может быть несколько значений приоритета: по умолчанию, текущий, максимально возможный. Информация, связанная с планированием: квант времени в течение, которого процесс выполняется при последовательном запуске. Информация о событии: идентификация события, инициализирующего продолжение работы события;

б) структуры данных: процесс может быть связан с другими процессами с помощью очереди, кольца или другими структурами. Для этого в управляемом блоке должны быть указатели на другие процессы;

в) обмен информацией между процессами: флаги, сигналы, сообщения;

г) привилегии процессов – это права доступа к определенным областям памяти, права выполнять определенные виды команд, возможности использования системных утилит;

д) управление памятью: это указатели на таблицы сегментов или страниц, описывающих распределение процессов в виртуальной памяти;

е) владение ресурсами и их использование: перечень открываемых файлов, сведения об исполнении процесса и других устройств ввода-вывода.

Память

Устройства

Файлы

Процессы

Таблицы

памяти

Таблицы ввода-вывода

Таблицы файлов

Процесс1

Процесс2

Процесс N

Процесс1

**Первичные**

**таблицы процессов**

**Образ процесса**

Рис. 3.2. Общая структура таблиц процесса с информацией по каждому

объекту управления

**Замечания по организации таблиц**

1. Хотя таблицы имеют разное назначение, все они должны быть связаны между собой и иметь перекрестные ссылки. Например, доступ к файлам, информация о которых хранится в таблице файлов, осуществляется через устройства ввода-вывода, а информация из файлов будет находиться в основной или виртуальной памяти на протяжении какого-то времени.

2. Для создания таблиц ОС должны иметь информацию по основной конфигурации вычислительной системы (например, такая информация, как объем основной памяти, количество и бит устройств ввода-вывода, идентификаторы устройств и т.д.) Сами таблицы должны быть доступны для ОС, поэтому место для них выделяется системой управления памяти.

***Таблицы памяти.***

Предназначены для хранения информации об основной ОП и виртуальной памяти. Эти таблицы включают следующую информацию:

1) объем основной памяти***,*** отведенной процессу;

2) объем вторичной или виртуальной памяти, отведенной процессу;

3) все атрибуты защиты блоков основной и виртуальной памяти;

4) всю информацию, необходимую для управления виртуальной памятью.

***Таблицы ввода-вывода.***

Используются для управления устройствами ввода-вывода и каналами компьютерной системы. В каждый момент времени устройство ввода-вывода может быть либо свободным, либо отданным в распоряжение какому-либо процессу. Если выполняется операция ввода-вывода, то должна быть информация о состоянии этой операции. Например, какие адреса ОП задействованы в этой операции, кто является отправителем и получателем отправляемой информации.

***Таблицы файлов.***

В них находится информация о существующих файлах, их расположение на магнитных носителях, текущем состоянии и других атрибутов. В ОС может быть специальная подсистема управления памятью.

***Таблицы процессов.***

Содержат сведения о процессах, располагая которыми ОС может управлять процессами.

**Управление процессами**

***Модели выполнения.***

Большинство процессов поддерживают два режима работы:

1. *Режим с невысокими привилегиями –* пользовательский.

2. *Режим с высокими привилегиями* – системный.

***Функции, возлагаемые на ядро ОС.***

1. Управление процессами:

а) создание и уничтожение;

б) планирование и диспетчеризация;

в) переключение;

г) синхронизация;

д) организация управления блокировкой.

2. Управление памятью:

а) выделение адреса пространства процесса;

б) свопинг;

в) управление страницами и сегментами.

3. Управление вводом-выводом:

а) управление буферами;

б) выделение процессам каналов и устройств ввода-вывода.

4. Функции поддержки:

а) обработка прерываний;

б) учет использования ресурсов;

в) текущий контроль системы.

***Создание процесса.***

При создании процесса необходимо выполнить следующие действия.

1. Присвоить началу процесса уникальный идентификатор.

2. Выделить память.

3. Инициализировать управляющий блок процесса.

4. Установить необходимые связи.

5. Создать или расширить другие структуры данных.

***Переключение процессов.***

Возникают вопросы.

1. Какие моменты должны приводить к переключению процессов?

2. Как установить разницу переключением режима работы и переключением процесса?

Переключение процесса может произойти в любой момент, когда управление от выполняющегося процесса переходит к системе.

Для это существуют следующие причины:

1. Прерывание, внешнее по отношению к текущей команде.

2. Ловушка, связанная с выполнением текущей команды.

3. Вызов супервизора, т.е. функции ОС.

Можно отметить несколько типов прерываний, которые приводят к переключению на систему:

1*. прерывание таймера*, когда процесс отработал свой квант.

2. *прерывание ввода-вывода*, когда процесс ждет ввода или уже вывел.

***Переключение режимов.***

Переключение режимов происходит при обработке прерываний. При этом процессор выполняет следующие действия:

1. Сохраняет контекст программы, устанавливает в счетчике команд адрес процедуры-обработчика прерывания.

2. Изменяет состояние процесса. При это выполняются следующие действия:

а) по сохранению полного контекста;

б) помещение управляющего блок процессов в соответствующую очередь;

в) выбор следующего для выполнения и восстановления контекста выбранного процесса.

**Концепция потока как составной части процесса**

Концепция процесса объединяет в себе две отдельные независимые концепции. Одна из них имеет отношение к владению ресурсами, а другая – к выполнению процесса. В некоторых ОС это привело к появлению конструкции, называемой *потоком*(thread).

В классическом понимании концепция процесса характеризуется двумя параметрами.

1*. Владение ресурсами* (resourceownership)

Процесс включает виртуальное адресное пространство, в котором содержится образ процесса, и на протяжении каких-то интервалов времени может владеть такими ресурсами, как основная память, каналы и устройства ввода/вывода, файлы, или на некоторое время получать контроль над ними. ОС выполняет защитные функции, предотвращая нежелательное взаимодействие процессов в сфере распределения ресурсов.

2. *Планирование выполнения*

Выполнение процесса осуществляется путём выполнения кода одной или нескольких программ, причём выполнение процессов может чередоваться друг с другом.

Процесс имеет такие параметры, как:

*–* состояние процесса;

*–* текущий приоритет, в соответствии с которым происходит планирование выполнения процесса.

Эти характеристики процесса являются его сущностью, но ОС может рассматривать их отдельно.

Для того, чтобы различать эти две характеристики, единицу диспетчеризации называют *потоком или облегченным процессом (lightweight process)*, а единицу владения ресурсами – *процессом или заданием.* Пока эта терминология не является полностью устоявшейся.

**Многопоточность**

*Многопоточность* – это способность ОС поддерживать в рамках одного процесса выполнение нескольких потоков. Процесс, представляющий собой единый поток выполнения, называется *однопоточным.*

*MS-DOS* является примером однопоточной ОС.

Процесс

Поток

Однопоточные ОС

*ОС UNIX* характеризуется следующей схемой

Процесс

Поток

*JAVA* является средой, в которой один процесс может расщепляться на несколько потоков.

*WINDOWS 2000, OS/2, LINUX* являются многопроцессорными, многопоточными системами.

В многопоточной среде процесс определяется, как структурная единица распределения ресурсов, а также как структурная единица защиты.

В рамках процесса может находиться один или несколько потоков, каждый из которых обладает следующими характеристиками:

1. Состояние выполнения потока

2. Сохраненный контекст невыполненного потока (поток имеет независимый счётчик команд, работающий в рамках процесса).

3. Стек выполнения.

4. Статическая память, выделяемая под переменные.

5. Доступ к памяти и ресурсам процесса, которому принадлежит поток. Этот доступ разделяется всеми потоками процесса.

**Однопоточная модель процесса**

Блок управления процессами

Адресное пространство пользователя

Стек пользователя

Стек

ядра

Рис. 3.3. Однопоточная модель процесса

В представление процесса с одним потоком входит:

- блок управления процесса;

- пользовательское адресное пространство;

- стек ядра;

- стек пользователя, с помощью которого осуществляется вызов процедур и возврат из них;

Когда выполнение процесса прерывается, то содержимое регистров сохраняется в памяти.

**Многопоточная модель процесса**

В многопоточной среде с каждым процессом также связаны управляющий блок и адресное пространство. Но для каждого потока создаётся свой отдельный стек и свой управляющий блок, в котором содержится значение регистров, приоритет и другая информация о состоянии потока.

Блок управления процессами

Адресное пространство пользователя

Стек пользователя

Стек

ядра

Стек пользователя

Стек

ядра

Стек пользователя

Стек

ядра

Управл. блок потока

Управл. блок потока

Управл. блок потока

Рис. 3.3. Многопоточная модель процесса

Все потоки процесса разделяют между собой состояние и ресурсы этого процесса. Они находятся в адресном пространстве и имеют доступ к данным. Если один поток изменяет какие-то данные, то другие потоки во время своего доступа к данным могут отследить эти изменения. Если один поток открывает файл, то другие потоки могут читать информацию из этого файла. Порождение новых потоков создаёт новую линию выполнения команд притом, что доступ идёт к одним и тем же данным.

*С точки зрения производительности, использование потоков имеет следующие преимущества.*

1. Создание нового потока в существующем процессе занимает на много меньше времени, чем создание нового процесса.

2. Поток можно завершить на много быстрее, чем процесс.

3. Переключение потоков в рамках одного процесса происходит на много быстрее, чем переключение процессов.

4. При использовании потоков повышается эффективность обмена информацией между двумя выполняющимися программами. В большинстве ОС обмен данными между независимыми процессами происходит с участием ядра. А так как потоки используют одно и тоже адресное пространство, то они могут обмениваться информацией без участия ядра.

Примером приложения, в котором можно применить поток, является файловый сервера. Так как серверу приходится обрабатывать большое количество запросов, то необходимо создание потоков. Если серверная программа работает на многопроцессорной машине, то на разных процессорах в рамках одного процесса могут выполняться одновременно несколько потоков.

**Функциональность потоков**

Основными состояниями потоков являются:

– состояние выполнения;

– состояние готовности;

– состояние блокировки.

С изменением состояния потоков связаны 4 действия.

*Порождение.*

Во время создания нового процесса, вместе с ним создаётся и поток этого процесса. В рамках одного и того же процесса поток может породить другой поток, определив его указатель команд и аргумент. Новый поток создаётся со своим собственным контекстом регистров и стековым пространством, после чего он помещается в очередь готовых к выполнению потоков.

*Блокирование.*

Если потоку нужно подождать, пока не наступит некоторое событие, то он блокируется. При этом сохраняется содержимое его пользовательских регистров, счётчик команд, указатель стека. После этого процессор может перейти к выполнению другого потока.

*Разблокирование.*

Когда наступает событие, которое ожидал блокированный поток, тогда этот поток переходит в состояние готовности.

*Завершение.*

После завершения стек потока удаляется.

Синхронизация потоков.

Так как все потоки используют одно и тоже адресное пространство и другие ресурсы, действия различных потоков надо синхронизировать, чтобы они не мешали друг другу и не повредили совместно используемые данные. При синхронизации потоков используются те же методы, что и при синхронизации процессов. Это значит, что решаются задачи потребитель-производитель, чтение-запись и другие.

**Взаимодействие процессов**

**.1 Задача взаимного исключения**

Если двум или более процессам необходимо взаимодействовать друг с другом, то они должны быть связаны, то есть иметь средства для обмена информацией. Предполагается, что процессы связаны слабо. Под этим подразумевается, что кроме достаточно редких моментов явной связи эти процессы рассматриваются как совершенно независимые друг от друга. В частности, не допускаются какие либо предположения об относительных скоростях различных процессов.

В качестве примера рассматривается два последовательных процесса, которые удобно считать циклическими. В каждом цикле выполнения процесса существует критический интервал. Это означает, что в любой момент времени только один процесс может находиться внутри своего критического интервала. Чтобы осуществить такое взаимное исключение, оба процесса имеют доступ к некоторому числу общих переменных. Операции проверки текущего значения такой общей переменной и присваивание нового значения общей переменной рассматриваются как неделимые. То есть, если два процесса осуществляют присваивание новое значение одной и той же общей переменной одновременно, то присваивание происходит друг за другом и окончательное значение переменной одному из присвоенных значений, но никак не их смеси. Если процесс поверяет значение переменной одновременно с присваиванием ей значения другим процессом, то первый процесс обнаруживает либо старое, либо новое значение, но никак не их смесь.

begin integer очередь;

очередь := 1;

parbegin

процесс 1: begin

L1: if (очередь = 2) then goto L1;

критический интервал 1;

очередь := 2;

остаток цикла 1;

goto L1;

end;

процесс 2: begin

L2: if (очередь = 1) then goto L2;

критический интервал 2;

очередь := 1;

остаток цикла 2;

goto L2;

end;

parend;

end;

int turn=1;

void P0()

{

while (1)

{

while(turn!=0)

критический интервал 1;

turn=1;

….

}

}

void P1()

{

while (1)

{

while(turn!=1)

критический интервал 2;

turn=0;

….

}

}

voidmain()

{

parbegin(P0,P1);

}

Недостатки решения:

1. Процессы могут входить в критический интервал строго последовательно. Темпы развития задаются медленным процессом.

2. Если кто-то из процессов останется в остатке цикла, то он затормозит и второй процесс

*Второй способ решения:*

begin integer С1,С2;

С1 := 1;

С2 := 1;

parbegin

процесс 1: begin

L1: if (С2 = 0) then goto L1;

С1 := 0;

критический интервал 1;

С1 := 1;

остаток цикла 1;

goto L1;

end;

процесс 2: begin

L2: if (С1 = 0) then goto L2;

С2 := 0;

критический интервал 2;

С2 := 1;

остаток цикла 2;

goto L2;

end;

parend;

end;

int flag[2];

void P0()

{

while (1)

{

while (flag[1]);

flag[0]=1;

критический интервал 1;

flag[0]=0;

….

}

}

void P1()

{

while (1)

{

while (flag[0]);

flag[1]=1;

критический интервал 2;

flag[1]=0;

….

}

}

voidmain()

{

flag[0]=0;

flag[1]=0;

parbegin(P0,P1);

}

Недостаток: принципиально при развитии процессов строго синхронно они могут одновременно войти в критический интервал.

*Было предложено следующее решение (вариант 3)::*

begin integer С1,С2;

С1 := 1;

С2 := 1;

parbegin

процесс 1: begin

А1: С1 := 0;

L1: if (С2 = 0) then goto L1;

критический интервал 1;

С1 := 1;

остаток цикла 1;

goto А1;

end;

процесс 2: begin

А2: С2 := 0;

L2: if (С1 = 0) then goto L2;

критический интервал 2;

С2 := 1;

остаток цикла 2;

goto А2;

end;

parend;

end;

int flag[2];

void P0()

{

while (1)

{

flag[0]=1;

while (flag[1]);

критический интервал 1;

flag[0]=0;

….

}

}

void P1()

{

while (1)

{

flag[1]=1;

while (flag[0]);

критический интервал 2;

flag[1]=0;

….

}

}

void main()

{

flag[0]=0;

flag[1]=0;

parbegin(P0,P1);

}

Недостаток: возникает другая проблема – может бесконечно долго решаться вопрос о том, кто первым войдет в критический интервал.

*Решение 4.*

begin integer С1,С2;

С1 := 1;

С2 := 1;

parbegin

процесс 1: begin

L1: С1 := 0;

if (С2 = 0) then begin

C1 := 1;

goto L1;

end;

критический интервал 1;

С1 := 1;

остаток цикла 1;

goto L1;

end;

процесс 2: begin

L2: С2 := 0;

if (С1 = 0) then begin

C2 := 1;

goto L2;

end;

критический интервал 2;

С2 := 1;

остаток цикла 2;

goto L2;

end;

parend;

end;

int flag[2];

void P0()

{

while (1)

{

flag[0]=1;

while (flag[1]);

{

flag[0]=0;

задержка;

flag[0]=1;

}

критический интервал 1;

flag[0]=0;

….

}

}

void P1()

{

while (1)

{

flag[1]=1;

while (flag[0]);

{

flag[1]=0;

задержка;

flag[1]=1;

}

критический интервал 2;

flag[1]=0;

….

}

}

void main()

{

flag[0]=0;

flag[1]=0;

parbegin(P0,P1);

}

*Решение задачи взаимного исключения. Алгоритм Деккера.*

begininteger С1,С2,очередь;

С1 := 1;

С2 := 1;

очередь := 1;

parbegin

процесс 1: begin

А1: С1 := 0;

L1: if (С2 = 0) then begin

if (очередь = 1) then goto L1;

C1 := 1;

B1: if (очередь = 2) then goto B1;

goto А1;

end;

критический интервал 1;

очередь := 2;

С1 := 1;

остаток цикла 1;

goto А1;

end;

процесс 2: begin

А2: С2 := 0;

L2: if (С1 = 0) then begin

if (очередь = 2) then goto L2;

C2 := 1;

B2: if (очередь = 1) then goto B2;

goto А2;

end;

критический интервал 2;

очередь := 1;

С2 := 1;

остаток цикла 2;

goto А2;

end;

parend;

end;

int flag[2], turn;

void P0()

{

while (1)

{

flag[0]=1;

while (flag[1]);

{

if (turn==1)

{

flag[0]=0;

while (turn==1);

flag[0]=1;

}

}

критический интервал 1;

turn=1;

flag[0]=0;

….

}

}

void P1()

{

while (1)

{

flag[1]=1;

while (flag[0]);

{

if (turn==0)

{

flag[1]=0;

while (turn==0);

flag[1]=1;

}

}

критический интервал 2;

turn=0;

flag[1]=0;

….

}

}

void main()

{

flag[0]=0;

flag[1]=0;

turn=1;

parbegin(P0,P1);

}

*Решение задачи взаимного исключения. Алгоритм Пэтерсона..*

int flag[2], turn;

void P0()

{

while (1)

{

flag[0]=1; turn=1;

while (flag[1] && (turn==1));

критический интервал 1;

flag[0]=0;

….

}

}

void P1()

{

while (1)

{

flag[1]=1; turn=0;

while (flag[0] && (turn==0));

критический интервал 2;

flag[1]=0;

….

}

}

voidmain()

{

flag[0]=0;

flag[1]=0;

parbegin(P0,P1);

}

Для доказательства корректности задачи взаимного исключения необходимо проверить три положения.

1. Решение безопасно в том смысле, что два процесса не могут одновременно оказаться в своих критических интервалах

2. В случае сомнения, кому из двух процессов первому войти в критический интервал, выяснение этого вопроса не откладывается до бесконечности

3. Остановка какого-либо из процессов в остатке цикла не вызывает блокировки другого процесса.

**.2 Обобщенная задача взаимного исключения**

Даны N процессов, каждый со своим критическим интервалом. Необходимо организовать их функционирование так, чтобы в любой момент самое большее один процесс находился в критическом интервале. Для проверки правильности решения проверяем три условия.

begin integer array b, c [0..N];

integer очередь;

for очередь = 1 step 1 until N do begin

b [очередь] := 1;

c [очередь] := 1;

end;

очередь := 0;

parbegin

процесс 1: begin … end;

процесс 2: begin … end;

…

процесс N: begin … end;

parend;

end;

процесс i: begin integer j, k;

Ai: b [i] := 0;

Li: if (очередь<> i) then begin

С[i] = 1;

k = очередь;

if (b [k] = 1) then очередь := i;

goto Li;

end;

C[i] := 0;

for j := 1 step 1 until N do

if ((j <> i) and (C[j] = 0)) then goto Li;

Критический интервал i;

очередь := 0;

C[i] := 1;

b[i] := 1;

Oстатокцикла i;

gotoAi;

end;

**Синхронизирующие примитивы (семафоры). Применение семафоров для решения задачи взаимного исключения**

Неделимое обращение к общей переменной всегда подразумевает одностороннее движение информации. Отдельный процесс может либо присвоить новое значение, либо проверить текущее значение. Однако такая проверка не оставляет следов для других процессов. Вследствие этого, в то время как процесс желает отреагировать на текущее значение общей переменной, значение этой переменной может быть изменено другими процессами. То есть такое взаимодействие через общие переменные нельзя считать во всех случаях адекватным. Для разрешения такой ситуации:

1. Вводятся специальные целочисленные общие переменные, называемые семафорами.

2. Добавляются к набору действия, из которых состоят процессы, два новых примитива: p-операция и v-операция.

Эти две операции всегда выполняются над семафорами и представляют единственный способ обращения к семафорам со стороны одновременно действующих процессов. Для решения задачи взаимного исключения область значений семафоров 0 и 1 (двоичные семафоры), но существует понятие и общего семафора, при котором он может принимать определенное количество целочисленных значений.

s1 – семафор,

p(s1) или v(s1) – запись операций над семафором.

V-операция – операция с одним аргументом, который должен быть семафором. Ее назначение – увеличение аргумента на единицу. Это действие рассматривается как неделимая операция.

Имеется достаточно принципиальное различие между семафорными и обычными операциями сложения. Если двумя параллельно-развивающимися процессами выполнена семафорная операция v(s1) над общим семафором s1, получим увеличение семафора s1 на два, так как операция v неделимая. А выполнение операции сложения s1 с единицой параллельными процессами может привести к увеличению значения s1 на единицу.

П1 П2

S1 := S1+1 S1 := S1+1

П1 П2

v(S1) v(S1)

P-операция – операция с одним аргументом, который должен быть семафором. Ее назначение – уменьшение аргумента на единицу, если только результирующее значение не становится отрицательным. Завершение р-операции, то есть решение о том, что в настоящий момент является подходящим для выполнения уменьшения и последующее уменьшение значения аргумента, рассматривается как неделимая операция.

P-операция определяет потенциальную задержку. Если инициируется р-операция над семафором, который в этот момент равен нулю, то в данном случае р-операция не может завершиться, пока какой-либо другой процесс не выполнит v-операцию над тем же семафором и не присвоит ему значение 1. Несколько процессов могут одновременно начать р-операцию над одним семафором.

Утверждение о том, что завершение р-операции есть неделимое действие, означает, что когда семафор получит значение 1, только одна из начавшихся р-операций над семафором завершится, какая именно – не определено.

begin integer свободно;

свободно := 1;

очередь := 0;

parbegin

процесс 1: begin … end;

процесс 2: begin … end;

…

процесс N: begin … end;

parend;

end;

процесс i:

begin

Li: p(свободно);

Критический интервал i;

V(свободно);

Остаток цикла i;

gotoLi;

end;

**Задача “производитель-потребитель”**

**Общие семафоры**

Рассматриваются два процесса, которые называются производитель и потребитель. Оба процесса являются циклическими. Производитель при каждом циклическом повторении участка программы производит отдельную порцию информации, которая должна быть обработана потребителем. Потребитель при каждом повторении обрабатывает следующую порцию информации, выработанную производителем. Отношения производитель-потребитель подразумевают односторонний канал связи, по которому могут передаваться порции информации. С этой целью процессы связаны через буфер неограниченной емкости. То есть, произведенные порции не должны немедленно потребляться, а могут организовывать в буфере очередь. Буфер работает по принципу FIFO.

ЧПБ – число порций в буфере.

РБ – работа с буфером

begin integer ЧПБ;

ЧПБ := 0;

parbegin

производитель: begin

П1: производство новой порции;

добавление новой порции в буфер;

v(ЧПБ);

goto П1;

end;

потребитель: begin

П2: p(ЧПБ);

взятие порции из буфера;

обработка взятой порции;

goto П2;

end;

parend;

end;

При неограниченном буфере производитель никаких ограничений не имеет, а потребитель может читать данные из буфера только если они там есть. Операции добавления порции к буферу и взятия порции из буфера могут помешать друг другу, если не предусмотреть их взаимное исключение во время исполнения. Этого можно избежать с помощью двоичного семафора.

begin integer ЧПБ, РБ;

ЧПБ := 0;

parbegin

производитель: begin

П1: производство новой порции;

p(РБ)

добавление новой порции в буфер;

v(PБ);

v(ЧПБ);

goto П1;

end;

потребитель: begin

П2: p(ЧПБ);

p(PБ);

взятие порции из буфера;

v(PБ);

обработка взятой порции;

goto П2;

end;

parend;

end;

При решении стремятся использовать только двоичные семафоры. В таком случае программа будет выглядеть так:

ЗП - задержка потребителя.

СЧПБ - счетчик порций в буфере.

begin integer ЧПБ, РБ, ЗП;

ЧПБ := 0;

РБ := 1;

ЗП := 0;

parbegin

производитель: begin

П1: производство новой порции;

p(РБ)

добавление новой порции в буфер;

ЧПБ := ЧПБ + 1;

If (ЧПБ = 1) then v(ЗП);

v(PБ);

goto П1;

end;

потребитель: begin integer СЧПБ;

ждать: p(ЗП);

продолжить: p(PБ);

взятие порции из буфера;

ЧПБ := ЧПБ - 1;

СЧПБ := ЧПБ;

v(PБ);

обработка взятой порции;

if (CЧПБ = 0) thengoto ждать;

else goto продолжить;

end;

parend;

end;

Значение локальной переменной СЧПБ устанавливается при взятии порции из буфера и фиксируется, не была ли эта порция последней. Здесь двоичный семафор РБ решает задачу взаимного исключения при работе с буфером. Введен новый двоичный семафор – задержка потребителя (ЗП). Программа представляет интерес в течение времени, когда буфер пуст. В исходном состоянии семафор ЗП установлен в ноль и открывается производителем только в том случае, когда в пустой буфер записывается порция. Остановка потребителя на семафоре ЗП не блокирует работу с буфером, так как стоит выше операции р(РБ). Смысл ждать есть только в том случае, если буфер пуст.

**Задача “производитель-потребитель”, буфер неограниченного размера**

begin

integer ЧПБ, РБ, ЗП;

ЧПБ:=0; РБ:=1; ЗП:=0;

parbegin

производитель: begin

n1: производство новой порции;

P(РБ);

добавление новой порции к буферу;

ЧПБ:=ЧПБ+1;

if (ЧПБ=0) then begin V(РБ); V(ЗП); end

else V(РБ);

goto n1;

end;

потребитель: begin

n2: P(РБ);

ЧПБ:=ЧПБ-1;

if (ЧПБ=-)1 then begin V(РБ); P(ЗП); P(РБ); end;

взятие порции из буфера;

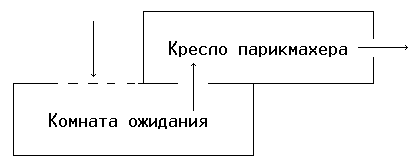
V(РБ);

обработка взятой порции;

goto n2;

end;

parend;



Эта программа называется "спящий парикмахер"

Используется два двоичных семафора: работа с буфером и задержка потребителя.

Особенности:

- используются только двоичные семафоры;

- задача взаимного исключения реализуется через семафоры;

- через семафоры реализуется задержка потребителя, которая действует при попытке потребителя читать данные из пустого буфера.

**Задача “производитель-потребитель”, буфер ограниченного размера**

Производитель и потребитель связаны через буфер ограниченной емкости в N порций. ЧПП - число пустых порций.

begin

integer ЧПБ, ЧПП, РБ;

ЧПБ:=0;

ЧПП:=N;

РБ:=1;

parbegin

производитель: begin

n1: производство новой порции;

P(ЧПП);

P(РБ);

добавление порции к буферу;

V(РБ);

V(ЧПБ);

goto n1;

end;

потребитель: begin

n2: P(ЧПБ);

P(РБ);

взятие порции из буфера;

V(РБ);

V(ЧПП);

обработка взятой порции;

goto n2;

end;

parend;

end;

И производитель и потребитель решают через РБ задачу взаимного исключения. Проблема производителя: нельзя писать в заполненный буфер. Проблема потребителя: нельзя читать из пустого буфера. Производитель решает свою проблему с использованием общего семафора ЧПП (число пустых порций), а потребитель ⎯ через ЧПБ (число порций в буфере).

**Взаимодействие через переменные состояния**

1.1 Постановка задачи

Исходные данные.

Можно определять и формировать исходные процессы.

Процессы могут связываться друг с другом через общие переменные.

Имеются средства синхронизации.

При взаимодействии процессов могут возникать решения, которые касаются более чем одного процесса. Не всегда очевидно, какое решение будет принято. Если не найден какой-то руководящий принцип (например, критерии эффективности), то с целью определённости нужно установить некоторые ограничения.

Если разрабатываются реально работающие системы, то требуется убедиться в корректности решения. При параллельном программировании использование проверяющих тестов затруднено, поэтому вопросы проверки правильности решения должны рассматриваться в самых начальных этапах проектирования системы.

1.2 Пример применения приоритетного правила

Рассматриваются производители, которые выдают порции информации различного размера и размер порции выражается в некоторых единицах. Потребитель обрабатывает последовательные порции из буфера и умеет обрабатывать порции, размер которых заранее не задан. Максимальный размер порции известен. Максимальный размер буфера определён в единицах информации, а не в количестве порций. Размер буфера - не менее максимального размера порции информации. Вопрос о возможности размещения в буфере определенной выработанной порции зависит от размера этой порции.

Если имеется более одного производителя и какой-то из них ждет из-за отсутствия достаточного места в буфере, то другие производители могут продолжать работу и достигнуть точки, когда они желают выдать выработанную порцию информации в буфер.

При принятии решения, кому первому поместить информацию в буфер, формируется требование: производитель, предлагающий большую порцию, имеет больший приоритет. Если порции равны, то не имеет значения, кто добавит информацию первым.

Дано: N ⎯ производителей; M ⎯ потребителей; RB ⎯ размер буфера.

begin

integer array желание[1:N], СП[1:N];

integer ЧПБ, ББ, РБ, ЧСЕБ, i;

for цикл:=1 step 1 until N do begin

желание[i]:=0;

СП[i]:=0;

end;

ЧПБ:=0; ЧСЕБ:=RB;

ББ:=0; РБ:=1;

parbegin

производитель 1: begin ... end;

. . . . . . . . . . . . . . . .

производительn: beginintegerРП;

цикл n: производство новой порции и установка размера порции (РП);

P(РБ);

if ( (ББ=0) and (ЧСЕБ>=РП) ) then

ЧСЕБ:=ЧСЕБ-РП

else

begin

ББ:=ББ+1;

желание[n]:=РП;

V(РБ);

P(СП[n]);

P(РБ);

end;

добавление порции к буферу;

V(РБ);

V(ЧПБ);

goto цикл n;

end;

... ... ...

производитель N: begin ... end;

потребитель 1: begin ... end;

... ... ...

потребитель m: begin

integer РП, i, max, nmax;

цикл m: P(ЧПБ);

P(РБ);

взятие порции из буфера и установка РП;

ЧСЕБ:=ЧСЕБ+РП;

проверка: if (ББ>0) then

begin

max:=0;

for i:=1 step 1 until N do

begin

if (max<желание[i]) then

begin

max:=желание[i];

nmax:=i;

end

end;

if (max=<ЧСЕБ) then

begin

ЧСЕБ:=ЧСЕБ-max;

желание[nmax]:=0;

ББ:=ББ-1;

V(СП[nmax]);

goto проверка;

end;

end;

V(РБ);

обработка взятой порции;

goto цикл m;

end;

Для решения задачи вводится переменная состояния для каждого производителя - “желание”, обозначающая число единиц информации в буфере, необходимых для размещения порции, которая в текущий момент не может быть добавлена к буферу. Если для процесса-производителя эта переменная равна нулю, то процесс-производитель не имеет неудовлетворённых требований на добавление порций.

Для каждого производителя вводится двоичный семафор СП (семафор производителя).

Для буфера вводится двоичный семафор РБ (работа с буфером), предназначенный для взаимного исключения при работе с буфером в широком смысле, т.е. не только взятие и добавление, но также проверка и модификация связанных с буфером переходных состояний.

Как только в буфер добавляется новая порция, она может быть обработана. Так как безразлично, кто из потребителей её возьмёт, то для определения этого может быть использован общий семафор ЧПБ (число порций в буфере). О свободных областях в буфере производителям сообщается через целочисленную переменную состояния ЧСЕБ (число свободных единиц в буфере). Введена целочисленная переменная ББ (блокировка буфера), значение которой определяет, сколько процессов-производителей имеют желание добавить порцию в буфер, но не смогли разместить свои порции в буфере и она уведомляет производителя в том, что уже есть процессы, которые обнаружили, что ББ>0, то он должен присоединиться к процессам, которые ожидают размещения порции в буфер.

**Монитороподобные средства синхронизации**

**Введение**

Основной концепцией таких средств является организация контроля правильности установления взаимосвязи между процессами.

Считается, что данные средства синхронизации имеют большую наглядность, чем семафорная техника, что достигается за счет структурирования.

Концепция монитороподобных средств основана на языковом обособлении или локализации средств взаимодействия. Это не только выделение особых языковых конструкций, но и сосредоточение в их составе информации о разделяемых ресурсах, о переменных состояния, характеризующих эти ресурсы, а также допустимые действия над этими ресурсами.

**Механизм типа «критическая область»**

Как следует из названия, это механизм, ориентированный на решение задачи взаимного исключения.

Для построения критической области используется две языковые конструкции. Одна из них VAR V: SHARED T; предназначена для описания критического ресурса, и её используют в начале текста программы в области описания типов переменных. Вторая описывает доступ к ресурсу в тексте программы REGION V DO S.

VAR R: SHARED T1;

Q: SHARED T2;

Begin

Parbegin

процесс 1: . . . L1: REGION R DO S1; . . .

процесс 2: . . . L2: REGION Q DO S2; . . .

. . . . . . . . . . .

процесс i: . . . Li: REGION R DO S3; . . .

Parend;

End.

Процессы 1-ый и i-ый взаимно разделяют ресурс R и при выполнении программы будет выполнятся только одна из областей.

Описание переменной V: SHARED T означает, что определяется ресурс с именем V, как некоторая переменная, доступная параллельным процессам. Тип ресурса задаётся его описанием T.

Для осуществления доступа к ресурсу V в тексте программы процесса требуется использовать конструкцию вида REGION V DO S. Такая конструкция описывает отдельную критическую область относительно критического ресурса V и определяет действия S, которые будут осуществлены над ресурсом. Такие конструкции при исполнении исключают друг друга относительно критического ресурса.

Основные допущения при построении такого механизма синхронизации заключаются в том, что недопустима обработка переменной, описанной как «разделяемая» в каких-либо языковых конструкциях в программе, отличных от конструкций типа REGION. Попытка сделать это – заведомая ошибка в использовании критического ресурса – может быть выявлена на этапе компиляции программы.

TYPE T = ARRAY 1..100 OF INTEGER;

VAR M: SHARED T;

Begin

Parbegin

Процесс 1: L1: <действие процесса>

REGION M DO <обработка массива M>;

GOTO L1;

. . . . . . . . . . . . . . . . . .

Parend;

End.

**Механизм типа «условная критическая область»**

TYPE T = ARRAY 1..100 OF INTEGER;

VAR M: SHARED T;

СЧИТЫВАНИЕ : BOOLEAN;

BEGIN

СЧИТЫВАНИЕ :=TRUE;

PARBEGIN процесс 1 : begin

m1: <некоторые действия>

region m do begin

await(считывание);

<считать информацию из М>

считывание := true;

end;

goto m1;

end;

процесс 2 : begin

m2: <некоторые действия>

region m do begin

<запись информации в М>

считывание := false;

end;

goto m2;

end;

parend;

end.

Данный механизм является модификацией механизма «критическая область». Но здесь вводится возможность производить работу с критическим ресурсом только тогда, когда он пребывает в определенном состоянии или выполнено условие, допускающее возможность работы с критическим ресурсом.

Для проверки условия и состояния вводится специальный примитив AWAIT. Он может выполняться только в пределах языковой конструкции REGION. В качестве параметра этого примитива используется логическое выражение. Предполагается, что значения компонентов логического выражения могут изменяться другими параллельными процессами, но обязательно в области REGION. Если при выполнении AWAIT окажется, что логическое выражение истинно, то данный процесс временно покидает критическую область. Это даёт возможность одному из других параллельных процессов войти в критическую область и изменить параметры логического выражения. При повторном входе приостановленного процесса в критическую область логическое условие может стать ложным, и процесс получит возможность работать с критическим ресурсом.

Для организации временного выхода из критической области и повторного входа в неё по отношению к критическому ресурсу выстраиваются две очереди:

- в первую или главную очередь попадают те процессы, которые готовы войти в критическую область, и конкурируют за право получить доступ к ресурсу;

- в другую очередь, называемую событийной, попадают процессы, которые, находясь в критической области, обнаружили с помощью примитива AWAIT невозможность обработки критического ресурса.

Процессы, находящиеся в событийной очереди, переносятся в главную очередь после каждого нормального выхода какого-либо из процессов из его критической области по отношению к какому-либо ресурсу.

Такие действия могут быть реализованы компилятором путём применения двоичного семафора, единственного в отношении каждого критического ресурса.

TYPE T = ARRAY 1..100 OF INTEGER;

VAR м : SHARED T; СЧИТЫВАНИЕ : BOOLEAN;

BEGIN

СЧИТЫВАНИЕ :=TRUE;

PARBEGIN

процесс 1 : begin

m1: <ДЕЙСТВИЯ ПРОЦЕССА> // процесс читать

reion m do begin await (считывание);

<считать информацию из М>

считывание := true;

end;

goto m1;

end;

процесс 2 :

begin

m2: <Действия процесса> //процесс-писатель

region m do begin

<запись информации в М>

считывание := false;

end;

goto m2;

end;

parend;

end.

Процесс-читатель может выполнить действия только в том случае, если информация в массив записана.

Особенностью данного средства синхронизации является сокрытие от пользователей механизма повторного входа в критические области. Этот механизм является механизмом активного ожидания, поэтому его использование целесообразно, когда обращение к критическим ресурсам происходит редко.

**Вторая модификация механизма «критическая область» (модификация второго рода)**

Сущность модификации второго рода состоит в том, что в составе функции REGION допустимо использование двух примитивов AWAIT и COUSE по отношению к переменной Е, называемой событийной.

Для описания этой переменной используется конструкция

VAR E : EVENT R.

Это значит, что Е – событийная по отношению к критическому ресурсу R.

Переменная Е является структурированной и представляет собой одномерный массив. В этот массив как в очередь попадают процессы, при работе которых был выполнен примитив AWAIT.

При выполнении примитива COUSE по отношению к этой переменной в ходе работы какого-либо из процессов все процессы, находящиеся в составе переменной E, получают возможность повторного входа в свои главные очереди.

При таком подходе процессы находятся в состоянии пассивного ожидания. Логическое условие явно не декларируется. О наступлении ожидаемого события сообщается с помощью примитива COUSE.

Пример.

Распределяется ресурс R, который состоит из M однотипных единиц. Любому процессу должна быть предоставлена возможность при каждом разовом обращении к системе получить из состава R одну единицу. Если в момент обращения нет ни одной свободной единицы, то процесс должен перейти в состояние ожидания до тех пор, пока не появится хотя бы одна свободная единица. После использования единицы ресурса процесс обязан её возвратить в состав ресурса R. О таком возвращении должны быть оповещены все процессы, ожидающие единицу ресурса.

BEGIN

TYPE RES = 1..M, P = 1..N;

VAR INF : SHARED RECORD

свобод\_ресурс : sequence of res;

требование : queue of p;

оч\_событий : array p of event r;

end;

procedure reserve(процесс : p; var ресурс : res)

begin

region inf do begin

while empty(свобод\_ресурс) do

begin

enter(процесс , требование);

await(оч\_событий(процесс));

end;

GET (ресурс, свобод\_ресурс);

end;

procedure release(ресурс : res)

begin

var процесс : р;

region inf do

begin

put(ресурс , свобод\_ресурс);

if (not empty(требование)) then

begin

remove(процесс , требование);

couse(оч\_событий(процесс));

end;

end;

end;

parbegin

процесс M1: begin

VAR R1:RES;

m1 : <некоторые действия>

Reserve(1 , var r1);

<использовать единицу ресурса из r с номером,

равным значению r1>

release(r1);

goto m1;

end;

. . .

процесс MN: begin

VARRN:RES;

m1 : <некоторые действия>

Reserve(N , varrN);

<использовать единицу ресурса из r с номером,

равным значению rN>

release(r1);

goto mN;

end;

parend;

end.

В рассматриваемом примере под критическим ресурсом понимается не ресурс R, который даже не описан, а некоторая совокупность информации, описывающая состояние ресурса и системы процессов, выдавших заявки на получение ресурсов. Для этого объявлена переменная INF, которую можно отнести к типу запись, состоящую из трёх поименованных элементов разного типа. Элемент с именем СВОБОД\_РЕСУРС – это переменная типа SEQUENCE (последовательность), состоящая из непоименованных элементов типа RES. Эта переменная предназначена для хранения информации о нераспределённых в текущий момент единицах ресурсов. Все единицы ресурса пронумерованы от 1 до M. Такие элементы можно обрабатывать с помощью примитивов GET и PUT.

GET присваивает значение переменной (1-ый аргумент) одному из элементов переменной типа SEQUENCE по правилу FIFO;

PUT – обратная по смыслу операции GET. При выполнении PUT по правилу FIFO выбирается элемент из переменной типа SEQUENCE и присваивается переменной, указанной первым аргументом.

Второе поле записи – ТРЕБОВАНИЕ. Это переменная типа QUEUE (очередь). Она содержит непоименованные элементы, каждый из которых может принимать значения от 1 до N, где N – максимальное число процессов, которые могут претендовать на использование ресурсов.

Элемент ТРЕБОВАНИЕ используется для хранения заявок, поступающих от процессов на приобретение единиц ресурсов. Заявка – это номер процесса. Все процессы пронумерованы от 1 до N, причём каждый процесс знает о своём номере. Занесение номера процесса в очередь и извлечение номера процесса из очереди осуществляется с помощью примитивов ENTER и REMOVE. Занесение и извлечение необязательно производятся по правилу FIFO, а в соответствие с некоторой дисциплиной обслуживания.

В отношении переменной ТРЕБОВАНИЕ используется процедура EMPTY, значение которой всегда будет истинным при отсутствии элементов в очереди ТРЕБОВАНИЕ.

Элемент ОЧ\_СОБЫТИЙ – это массив событийных переменных, каждая из которых указывает явно на один и тот же ресурс. Процессу с номером i соответствует i-ый элемент этого массива.

Для упрощения решения задачи используются процедуры RESERVE (для выделения единицы ресурса) и RELEASE (для возврата использованного ресурса в переменную СВОБОД\_РЕСУРС).

При обращении к процедуре RESERVE процесс передаёт в качестве параметра собственный номер и имя переменной, куда будет помещён номер единицы ресурса, выделенной для использования.

Процедура освобождения RELEASE требует один параметр, и в этом параметре указывается номер той единицы ресурса, которая возвращается процессом в состав общего ресурса. Предполагается, что процесс может возвращать только те единицы ресурса, которые он захватил.

В каждой процедуре имеется критическая область в отношении ресурса INF, и только в её пределах происходит обращение к ресурсам.

При необходимости получить ресурс процесс с соответствующим номером обращается к процедуре RESERVE, причём такое обращение может сделать только один их процессов. Остальные по правилу взаимного исключения переводятся в состояние ожидания относительно критического ресурса INF.

Если процесс, который вошёл в критическую область, при выполнении процедуры резервирования обнаружил, что ни одной свободной единицы ресурса нет, то он временно покидает критическую область с помощью примитива AWAIT и переходит в состояние ожидания на собственной событийной переменной в массиве ОЧ\_СОБЫТИЙ.

Если процесс, находясь в критической области, обнаружил хотя бы одну свободную единицу ресурса, он узнаёт её номер и производит её захват. При этом корректируется значение переменной СВОБОД\_РЕСУРС. Полученной единицей ресурса процесс владеет монопольно, после чего осуществляется выход из критической области, и, следовательно, в неё может войти какой-либо из других процессов.

При выполнении процессом процедуры освобождения указанная им единица ресурса становится свободной, и соответствующая информация заносится в переменную СВОБОД\_РЕСУРС. В ходе выполнения этой процедуры также проверяется необходимость реактивации процессов, которые временно вышли из критической области. Эта проверка проводится путём анализа переменной ТРЕБОВАНИЕ, и если в этой переменной имеются запросы, то осуществляется перевод одного из процессов из переменной ОЧ\_СОБЫТИЙ в главную очередь.

Предложенный способ обращения к критическому ресурсу путём использования процедур в тексте алгоритма является методологическим приёмом, который позволяет назвать такие средства синхронизации монитороподобными. Здесь увеличена степень локализации действий относительно критического ресурса INF. Такой подход позволяет разделить процесс проектирования алгоритма на два класса. В одном проектируются средства доступа к ресурсам или система распределения ресурса, а второй – это непосредственно разработка программы, реализующей определённую задачу.

Таким образом, создаётся централизованная схема распределения критических ресурсов. Такая схема называется монитороподобной.

***Тема 8. Ресурсы***

**Распределение ресурсов. Проблема тупиков**

Рассматривается логическая задача, которая возникает при взаимодействии различных процессов, когда они должны делить ресурсы.

Под процессами понимаются программы, описывающие некоторый вычислительный процесс, выполняемый ЭВМ. Выполнение такого вычислительного процесса требует времени, в течение которого в памяти ЭВМ хранится информация.

О процессах известно следующее.

1. Их требования к объему памяти не будут превышать определенного предела.

2. Каждый вычислительный процесс завершится при условии, что требуемый процессу объем памяти будет предоставлен в его распоряжение. Завершение вычислительного процесса будет означать, что его требование к памяти уменьшилось до нуля.

Имеющаяся память поделена на страницы фиксированного размера, эквивалентные с точки зрения программы.

Фактическое требование нужного процессу объема памяти является функцией времени, то есть изменяется по мере протекания процесса, но не превышает заранее заданную границу. Отдельные процессы запрашивают и выделяют память единицами в одну страницу. Однажды начатый процесс получает возможность рано или поздно завершиться, и исключается ситуация, когда процесс может быть уничтожен в ходе выполнения, напрасно истратив свои ресурсы.

Если на вычислительной машине выполняется один процесс, или они последовательно следуют один за другим, единственным условием является то, чтобы запрашиваемый процессом объём памяти не превышал доступный объём памяти вычислительной машины. Если параллельно развиваются несколько процессов, то могут возникнуть проблемы с выделением им ресурсов и надо предусмотреть выделение памяти таким образом, чтобы все начатые процессы смогли завершить свое выполнение.

Ситуация, когда какой-либо из процессов может быть завершён лишь при условии уничтожения какого-либо другого процесса, называется «смертельными объятиями» или тупиком.

Рис. 4.1. Тупиковая ситуация

Ситуация, приведенная выше, является тупиковой. Из этой ситуации невозможно выйти без уничтожения какого-либо из процессов.

Решаемая проблема состоит в том, как избежать попадания в тупик, не накладывая слишком больших ограничений.

**Алгоритм банкира**

Банкир обладает конечным капиталом, например, талерами. Он решает принимать клиентов, которые могут занимать у него талеры на следующих условиях:

1. Клиент делает заем для совершения сделки, которая будет завершена за определенный промежуток времени.

2. Клиент должен указать максимальное количество талеров для этой сделки.

3. Пока заем не превысит заранее установленную потребность, клиент может увеличивать или уменьшать свой заем.

4. Клиент, который просит увеличить свой текущий заем, без недовольства воспринимает ответ о том, что необходимо подождать с получением очередного талера, но через некоторое время талер будет обязательно выдан.

5. Гарантия для клиента, что такой момент наступит, основана на предусмотрительности банкира и на том факте, что остальные клиенты работают по таким же правилам.

Основными вопросами при решении такой задачи являются:

1. При каких условиях банкир может заключить контракт с новым клиентом?

2. При каких условиях банкир может выплатить (следующий) запрашиваемый талер клиенту, не опасаясь попасть в тупик?

Ответ на первый вопрос достаточно прост: банкир может принять любого клиента на обслуживание, чья максимальная потребность не превышает капитал банкира. Ответ на второй вопрос достаточно сложный. Сначала нужно формализовать задачу.

потребность[i] <= капитал, для всех i.

0 <= заем[i] <= потребность[i], для всех i.

требование[i] = потребность[i] - заем[i], для всех i.

наличные = капитал - СУММА\_по\_i заем[i].

0 <= наличные <= капитал.

В такой ситуации алгоритм принятия решения о том, будет ли безопасной выдача следующего талера выглядит след образом:

integer Св\_Деньги; boolean Безопасно;

boolean array Завершение\_под\_сомнением [1..N];

Св\_Деньги := наличные;

for i := 1 step 1 until N do Завершение\_под\_сомнением[i] := true;

L: for i :=1 step 1 until N do begin

if ( (Завершение\_под\_сомнением [i]) and

(Требован[i] <= Св\_Деньги) ) then begin

Завершение\_под\_сомнением [i] := false;

Св\_Деньги := Cв\_Деньги + Заем[i];

goto L;

end;

end;

if (Св\_Деньги = капитал) then Безопасно := true

else Безопасно := false;

Проверка возможности выплаты, то есть положение Безопасно, означает, могут ли с гарантией быть завершены все сделки. Алгоритм начинается с проверки, имеет ли, по крайней мере, один клиент требование, не превышающее наличные деньги. Если это так, то этот клиент может завершить свою сделку, и далее исследуются оставшиеся клиенты с учетом того, что первый клиент завершил свою сделку и возвратил свой заём полностью.

Безопасность положения означает, что могут быть закончены все сделки, то есть банкир видит способ получения обратно всех своих денег.

В полном варианте алгоритма банкира эта ситуация должна быть удовлетворена для всех клиентов, принятых на обслуживание и если после завершения цикла, отмеченного меткой L окажется, что капитал банкира полностью восстановлен, то ситуация считается безопасной, в противном случае она определяется как тупиковая и, следовательно, удовлетворять запрос клиента не представляется возможным.

Если более глубоко анализировать эту проблему, то можно показать, что решение о выделении следующего запрашиваемого талера клиенту будет принято тогда, когда хотя бы для одного клиента выполниться условие ((Завершение\_под\_сомнением[i]) and (Треб[i]<=Св\_Деньги)) и это говорит о безопасности ситуации и проверку можно приостановить.

**Применение алгоритма банкира**

Каждому талеру можно поставить в соответствие жесткий диск или какое-либо устройство. Тогда заем талера будет означать разрешение на использование одного из дисков.

begin

integer array Заем, Требование, Клиент\_Сем,

Клиент\_Пер, Номер\_Талера,

Возвращенные\_Талеры[1..N];

integer array Талер\_Сем, Талер\_Пер, Номер\_Клиента[1..M];

integer Взаимн\_искл, наличные, k;

boolean procedure Попытка\_выдать\_талер\_клиенту (integer j);

begin

if (Клиент\_Пер[j]=1) then begin

integer i, Своб\_Деньги;

boolean array Заверш\_под\_сомн[1..N];

Своб\_Деньги :=наличные -1;

Требование[j]:=Требование[j]-1;

Заем[j]:=Заем[j]+1;

for i:=1 step 1 until N do

Завершен\_под\_сомн[j]:=true;

L0: for i:=1 step 1 until N do begin

if (Завершен\_под\_сомн[j] and

(Требование[i]=<Своб\_Деньги) ) then begin

if (i<>j) then begin

Заверш\_под\_сомн[i]:=false;

Своб\_Деньги:=Своб\_Деньги+Заем[i];

goto L0;

end;

else begin

i:=0;

L1: i:=i+1;

if (Талер\_Пер[i] = 0) then goto L1;

Номер\_Талера[j]:=i;

Номер\_Клиента[i]:=j;

Клиент\_Пер[j]:=0;

Талер\_Пер[i]:=0;

Наличные:=Наличные-1;

Попытка\_Выдать\_Талер\_Клиенту:=true;

V (Клиент\_Сем[j]);

V (Талер\_Сем[j]);

goto L2;

end;

end;

end;

Требование[j]:=Требование[j]+1;

Заем[j]:=Заем[j]-1;

end;

Попытка\_Выдать\_Талер\_Клиенту:=false;

L2: end; /\* процедуры \*/

Взаимн\_искл:=1; Наличные :=M;

for k:=1 step 1 until N do begin

Заем[k]:=0;

Клиент\_Сем[k]:=0;

Клиент\_Пер[k]:=0;

Требование[k]:=Потребность[k];

Возвращенные\_Талеры[k]:=Потребность[k];

end;

for k:=1 step 1 until M do begin

Талер\_Сем[k]:=0;

Талер\_Перем[k]:=1;

end;

parbegin

Клиент 1: begin ... end;

…

Клиент i: begin ...

P(Возвращенные\_Талеры[i]);

P(Взаимн\_искл);

Клиент\_Пер[i] := 1;

Попытка\_выдать\_талер\_клиенту(i);

V(Взаимн\_искл);

P(Клиент\_Сем[i]);

…

end;

Талер 1: begin ... end;

…

Талер m: begin integer h;

…

начало: P(Талер\_Сем[m]);

P(Взаимн\_искл);

Требование[Номер\_Клиента[m]] := Требование[Номер\_Клиента[m]] - 1;

Талер\_Перем[m] := 1;

наличные := наличные + 1;

V(Возвращенные\_Талеры[Номер\_Клиента[m]]);

for h:=1 step 1 until N do begin

if (Попытка\_выдать\_талер\_клиенту(h))

then goto выход;

end;

выход:V(Взаимн\_искл);

goto начало

…

end;

parend;

end;

Сущность: есть клиент и есть талер. Каждый клиент имеет переменную состояния Клиент\_Пер. Если Клиент\_Пер = 1 – это означает: “хочу получить заем”. Иначе Клиент\_Пер = 0. Каждый талер имеет переменную состояния Талер\_Пер. Если Талер\_Пер = 1 – это означает: “нахожусь среди свободного капитала”. Клиенты пронумерованы от 1 до N, а талеры от 1 до M. С каждым клиентом связывается переменная Номер\_Талера, значение которой после очередной выдачи талера клиенту определяет номер только что выделенного талера. В свою очередь с каждым талером связана переменная Номер\_Клиента , значение которой указывает клиента, которому выдан этот талер. Имеются семафоры Клиент\_Сем, и Талер\_Сем. Фактически возврат талера заканчивается после того, как тот действительно присоединится к наличному капиталу банкира: об этом талер будет сообщать клиенту с помощью общего семафора клиента "Возвращенные\_талеры". Значение булевой процедуры "Попытка\_выдать\_талер\_клиенту" говорит о том, удовлетворен ли задержанный запрос на талер. В программе для талера используется тот факт, что возврат талера может теперь привести к удовлетворению единственного задержанного запроса на талер (если банкир распоряжается услугами более чем одного вида, то последнее свойство уже не будет выполняться).

***Тема 9. Память. Управление памятью***

**Требования к управлению памятью**

Система управления памятью должна обеспечивать решение следующих проблем.

1. Перемещение. Для максимальной загрузки процессора необходимо иметь набор готовых к выполнению процессов. Для этого требуется реализовать возможные загрузки и выгрузки активных процессов из основной памяти и обратно. При этом весьма желательно, чтобы при таких перезагрузках программа могла перемещаться в различные участки памяти. ОС должна знать местоположение управляющей информации процессов и стека выполнения, а также точку входа в исполняемый код. Т.к. управлением памятью занимается ОС, а также ОС управляет загрузкой процессов в память, соответствующие адреса она получает автоматически. Однако в программе встречаются команды перехода и ветвления, и поэтому должна быть реализована возможность преобразования адресных ссылок в коде программы в реальные физические адреса, соответствующие текущему положению программы в ОП.
2. Защита. Каждый процесс должен быть защищен от воздействия других процессов. Это означает, что из программного кода одного процесса нельзя обращаться к памяти, отведенной под другой процесс. Но обеспечение перемещаемости программы усложняет организацию механизма защиты. Т.к. в программе невозможно предусмотреть контроль за всеми адресными обращениями, то такой контроль возможен только на аппаратном уровне.
3. Совместное использование. Предусматривает возможность обращения нескольких процессов к ОП. Если несколько процессов выполняют один и тот же машинный код, то целесообразно разрешить каждому из процессов работать с одной и той же копией этого кода, а не создавать каждому процессу копию. Система управления памятью должна обеспечить управляемый доступ к разделенным областям памяти, не ослабляя при этом защиту памяти.
4. Логическая организация памяти. И основная память и вторичная (внешняя) организованы как линейное адресное пространство, где элементом адресации является байт или слово. Такая организация отражает систему аппаратного обеспечения, но не соответствует страничной организации программ. Большинство программ организуется из модулей, часть из которых неизменна (используются только для чтения или исполнения), а другие содержат данные, которые могут изменяться.

Если ОС или аппаратное обеспечение ЭВМ могут работать с программами, представленными в виде модулей, то это обеспечивает ряд преимуществ:

- модули могут быть созданы и откомпилированы независимо друг от друга, при этом ссылки из одного модуля во второй разрешается системой во время работы программы;

- разные модули могут получать разные модули защиты;

- возможно применение механизмов совместного использования модулей различными процессами.

Наиболее подходящим способом для решения таких задач является сегментация.

5. Физическая организация памяти. Память в ЭВМ разделяется как минимум на два уровня- основную и вторичную. Основная обеспечивает быстрый доступ по достаточно высокой цене, она энергозависима, следовательно, не обеспечивает долговременного хранения. Вторичная память медленнее и дешевле, и энергонезависима. Следовательно, она может использоваться для долговременного хранения данных и программ. Основная память применяется для хранения программ, используемых в текущее время.

Одной из основных задач ОС является организация потоков информации между основной и вторичной памятью.

**Схемы распределения памяти**

Основной функцией ОС по управлению памятью является размещение программы в основной памяти для её выполнения процессором. В современных ОС решение этой задачи предполагает использование сложной схемы, называемой ***виртуальной памятью***.

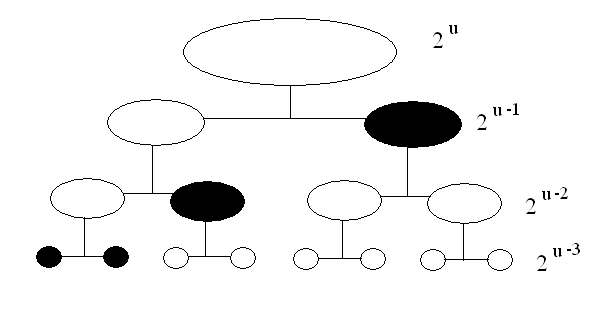
Известно несколько способов распределения памяти:

1. фиксированное распределение. ОП разделяется на ряд статических разделов во время генерации системы. Процесс может быть загружен в раздел равного или большего размера. Положительная сторона- простота реализации и малые системные затраты. Отрицательная сторона – неэффективное использование памяти из-за внутренней фрагментации и фиксированного максимального количества процессов.
2. Динамическое распределение. Разделы создаются динамически, каждый процесс загружается в раздел необходимого раздела. Достоинство- отсутствие внутренней фрагментации, более эффективное использование ОП. Недостаток – существенные затраты процессора на противодействие внешней фрагментации и проведения уплотнения памяти. При выделении памяти, таким образом, применяются три основных алгоритма: наилучший подходящий, первый подходящий, следующий подходящий.
3. Простая страничная организация. ОП разделена на ряд кадров равного размера. Каждый процесс распределен на некоторое количество страниц равного размера, такой же длины, что и кадры памяти. Процесс загружается путем загрузки всех его страниц. Достоинство- отсутствие внешней фрагментации. Недостаток – небольшая внутренняя фрагментация.
4. Простая сегментация. Каждый процесс распределен на ряд сегментов. Процесс загружается путем загрузки всех своих сегментов в динамические, не обязательно смежные, разделы. Достоинство- отсутствие внутренней фрагментации. Недостаток – проблемы с внешней фрагментацией.
5. Страничная организация виртуальной памяти. Подобна простой страничной организации, но не требуется загружать все страницы процесса. Необходимые нерезидентные страницы автоматически подгружаются в память. Достоинства – отсутствие внешней фрагментации, более высокая степень многозадачности, большое виртуальное адресное пространство. Недостаток – значительные затраты на управление виртуальной памятью.
6. Сегментация виртуальной памяти. Подобна простой сегментации, но не требуется загружать все сегменты процесса. Необходимые нерезидентные сегменты автоматически подгружаются в память. Достоинства - отсутствие внутренней фрагментации, более высокая степень многозадачности, большое виртуальное адресное пространство, поддержка защиты и совместного использования. Недостаток – затраты на управление сложной виртуальной памятью.

**Система двойников при распределении памяти**

Фиксированное распределение памяти ограничивает количество активных процессов и неэффективно использует память при несоответствии между размерами разделов и процессами. Динамическое распределение реализуется более сложно и включает затраты на уплотнение памяти. Решением в этом плане является ***система двойников***. В ней память распределяется блоками размером , где -минимальный размер выделяемого блока, а -максимальный размер(вся доступная распределенная память). Вначале все доступное для распределения адресное пространство рассматривается как единый блок размером .

Если запрашивается блок размером , таким что , то выделяется весь блок памяти, в противном случае, блок будет разделен на два одинаковых подблока (двойника), размерами. Если , то выделяется блок, иначе повторяется. Процесс деления продолжается до тех пор, пока не будет сгенерирован наименьший блок, размер которого не меньше. Система двойников всегда ведет список доступных блоков для каждого размера где . Блок может быть удален из списка путем разделения его пополам и внесения двух свободных блоков в список . Когда пара свободных блоков в списке  оказывается освобожденной, они удаляются из этого списка и объединяются в единый блок в списке .



На каждом уровне введен список свободных и занятых областей памяти.

***Тема 10. Организация виртуальной памяти***

**Структуризация адресного пространства виртуальной памяти**

Сложность управления виртуальной памятью объясняется необходимостью устанавливать связь между аппаратурой процессора и программным обеспечением ОС. Ключевыми моментами в таком управлении являются следующие характеристики страничной организации и сегментации.

1. Все обращения к памяти в рамках процесса представляют собой логические адреса, которые динамически транслируются в физические во время выполнения. Процесс может находиться в разных местах оперативной памяти (ОП). Логический адрес представляет собой ссылку на ячейку памяти, не зависимо от расположения данных в памяти. Относительный адрес определяется положением от некоторой известной точки, обычно начала программы, и представляет собой частный случай логического адреса. Физический адрес представляет собой действительное расположение ячейки в ОП.
2. Процесс может быть разбит на ряд частей, которые могут не располагаться в ОП единым непрерывным блоком. Это обеспечивается за счет динамической трансляции адресов и использования таблицы страниц или сегментов.

В результате этого достигаются следующие свойства:

– в ОП может поддерживаться (выполняться) большое количество процессов;

– процесс может быть больше, чем вся ОП.

Процесс выполняется в ОП, которая поэтому называется реальной, но программист имеет дело с потенциально гораздо большей памятью, называемой виртуальной и которая выделяется во внешней памяти. При управлении виртуальной памятью решаются 4 задачи:

- размещение;

- перемещение;

- преобразование;

- замещение.

**Задачи управления виртуальной памятью**

**Задача размещения**

Её сущность состоит в выборе в адресном пространстве ОП сегментов или страниц, на которые будут отображаться сегменты или страницы виртуального адресного пространства.

При решении этой задачи стараются выбрать максимально простой алгоритм распределения памяти. Алгоритм строится таким образом, чтобы уменьшить частоту его использования. Для учета свободных и распределенных страниц можно использовать двоичный вектор. Число двоичных разрядов вектора равно числу страниц ОП. Двоичные разряды нумеруются в той же последовательности, что и страницы. Если в разряде храниться 0 - это значит, что страница свободна и 1 - страница распределена. Если свободных страниц нет, а запрос на страницу поступает, то решается задача замещения и какую-то страницу перемещают во внешнюю память.

**Задача перемещения**

Состоит в том, как произвести выборку информации, которая хранится во внешней виртуальной памяти, для переноса в ОП. Используется 2 основных варианта выборки информации во внешней памяти:

– по требованию;

– предварительно.

При выборке по требованию страница передается в ОП только тогда, когда выполняется обращение к ячейке памяти, расположенной на этой странице. Когда процесс только запускается, число обращений к внешним страницам, распределенных во внешней памяти, достаточно велико, но постепенно начинает срабатывать принцип локализации, и все большее число обращений начинает происходить к уж загруженным страницам.

В случае предварительной выборки, загружается не только страница, вызвавшая прерывание. Если страница процесса расположена последовательно во внешней памяти, то бывает эффективной загрузка нескольких страниц за один раз. Предварительная выборка ориентирована на физическую организацию внешней памяти, и здесь учитываются такие факторы, как время поиска страницы и задержки, связанные с позиционированием устройств чтения.

**Задача преобразования**

Заключается в нахождении абсолютного физического адреса основной памяти по его виртуальному адрес

Самый простой способ – это одноуровневая таблица. Ее недостатком является большой размер страниц.

|  |  |
| --- | --- |
| Лог. адрес | Физ. адрес |
| 1 | 100 |
| 2 | 70 |
| 3 | 215 |
| 4 | 23 |
| 5 | 47 |
|  |  |

Поэтому используют двухуровневые таблицы, где изначально реализуется каталог страниц, а потом уже и таблицы страниц.

**Задача замещения**

Целью этой задачи является выбор среди пространства ОП той страницы, которую следует переместить во внешнюю память. Задача решается, когда обнаруживается отсутствие свободной страницы в ОП, а вместе с тем следует запрос на размещение новых страниц.

Первый, наиболее простой подход, заключается в остановке процесса, потребовавшего страницу ОП в ситуации, когда вся память распределена. В этом случае управление передается другому процессов, у которого нет требований на дополнительную память, а в отношении приостановленного процесса либо не принимается никаких действий, либо все его страницы выгружаются во внешнюю память. Недостаток этого подхода- дискриминации подвергается тот процесс, в ходе выполнения которого возникла потребность замещения.

Идеальная стратегия замещения: должна быть замещена та страница, к которой дольше всего не будет обращений в будущем.

Существуют следующие стратегии:

- для замещения выбирается страница случайным образом;

- выбирается страница, которая дольше всего была в ОП;

- FIFO (удаляется та страница, которая раньше всех была распределена какому-либо процессу). Фактически, это реализация предыдущей стратегии, но она может быть реализована для различных процессов;

- алгоритм удаления дольше всех неиспользовавшейся страницы. Если долго обращения к странице не было, следовательно, в будущем тоже не предвидится.

В простейшем случае, с каждой страницей для этой стратегии связывается бит использования. Этот бит установлен в 1 при обращении к странице, а способ сброса бита в 0 и определяет способы реализации данной стратегии.

В настоящее время для решения задачи замещения находят применение варианты «часового» алгоритма.

Страница 1

Use 1

Страница 45

Use 1

Страница 47

Use 1

Страница53

Use 0

Use 0

Страница 108

…

…

Указатель буфера

Рис. 6.1. Иллюстрация “Часового” алгоритма

Имеется циклический буфер размерности n, в каждом элементе хранится номер страницы и бит использования. Бит использования устанавливается в 1, когда к странице произведено обращение, этот бит также устанавливается при первой загрузке страницы в ОП. Указатель буфера указывает на последнюю замещенную страницу. Когда возникает потребность решить задачу замещения, указатель перемещается на следующий элемент буфера. Если бит использования установлен в 0, то производится замещение соответствующей страницы. Если же окажется, что бит использования равен 1, то страница не замещается, бит использования устанавливается в 0, а указатель перемещается на следующий элемент буфера. Перемещение указателя осуществляется до тех пор, пока не будет обнаружена страница с 0 (нулевым) битом использования.

Повысить эффективность часового алгоритма можно путем увеличения количества используемых при его работе битов. Практически, во всех системах страничной организации со страницей связывается бит модификации. Этот бит указывает, что страница не может быть замещена до тех пор, пока её содержимое не будет переписано во внешнюю память. Соответственно может быть 4 комбинации битов использования и битов модификации:

*nm*

*0 n=0-давно использован*

*0 1-недавно использован*

*1 m=0- не модифицирован*

*1 1- модифицирован*

Часовой алгоритм выглядит следующим образом:

1. Сканируем буфер, начиная с текущего положения. В процессе сканирования бит использования не изменяется. Первая страница с состоянием битов (0,0) замещается.
2. Если такой страницы нет, то ищем страницу с параметрами (0,1). Если такая страница найдена, она замещается. В процессе выполнения данного шага у всех просмотренных страниц сбрасывается бит использования.
3. Если выполнение шага 2 не дало результата, значит, у всех страниц будет сброшен бит использования, указатель буфера вернется в исходное положение, затем повторяем шаг 1 и , при необходимости, 2.

Явление пробуксовки наблюдается тогда, когда ОП имеет небольшие размеры, а программы велики по размеру. В этом случае может возникнуть ситуация частого замещения страниц, и большая часть процессорного времени тратится на выполнение служебных функций. Следовательно, резко замедляется выполнение пользовательских программ.

***Тема 11. Планирование в операционных системах***

Вмногозадачных системах в основной памяти одновременно содержится код нескольких процессов. В работе каждого процесса периоды использования процессора чередуются с ожиданием завершения выполнения операций ввода-вывода или некоторых внешних событий. Процессор (или процессоры) занят выполнением одного процесса, в то время как остальные находятся в состоянии ожидания.

Ключом к многозадачности является планирование. Обычно используются четыре типа планирования (табл. 7.1)

Таблица 7.1 - Типы планирования

|  |  |
| --- | --- |
| Долгосрочное планирование | Решение о добавлении процесса в пул выполняемых процессов |
| Среднесрочное планирование | Решение о добавлении процесса к числу процессов, полностью или частично размещённых в памяти |
| Краткосрочное планирование | Решение о том, какой из доступных процессов будет выполнятся процессором |
| Планирование ввода-вывода | Решение о том, какой из запросов процессоров на операции ввода-вывода будет обработан свободным устройством ввода-вывода |

**Типы планирования процессора**

Цель планирования процессора состоит в распределении во времени процессов, выполняемых процессором (или процессорами) таким образом, чтобы удовлетворять требованиям системы, таким, как время отклика, пропускная способность и эффективность работы процессора. Во многих системах планирование разбивается на три отдельные функции — долгосрочного, среднесрочного краткосрочного планирования. Их названия соответствуют временным масштабам выполнения этих функций.

На рис. 7.1 функции планирования привязаны к диаграмме переходов стояния процесса. Долгосрочное планирование осуществляется при создании нового процесса и представляет собой решение о добавлении нового процесса к множеству активных в настоящий момент процессов. Среднесрочное планирование является частью свопинга и представляет собой решение о добавлении процесса к множеству по крайней мере частично расположенных в основной памяти (и, следовательно, доступных для выполнения) процессов. Краткосрочное планирование является решением о том, какой из готовых к выполнению процессов будет выполняться следующим. На рис. 7.2 диаграмма переходов реорганизовала таким образом, чтобы показать вложенность функций планирования.

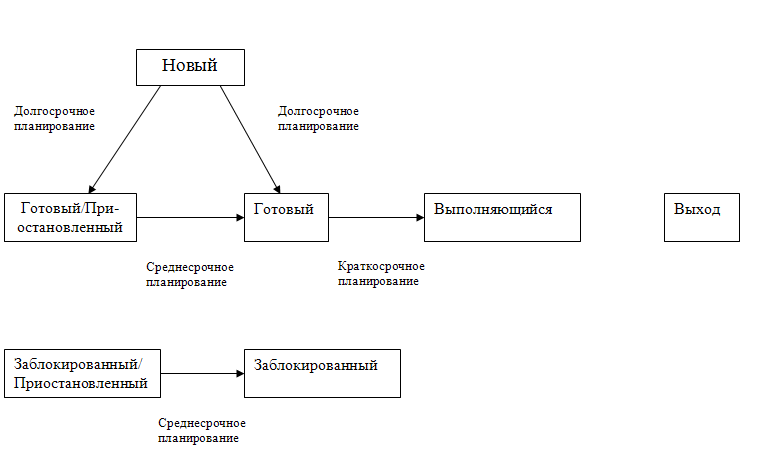


Рис. 7.1 Место планирования в диаграмме переходов состояний процесса

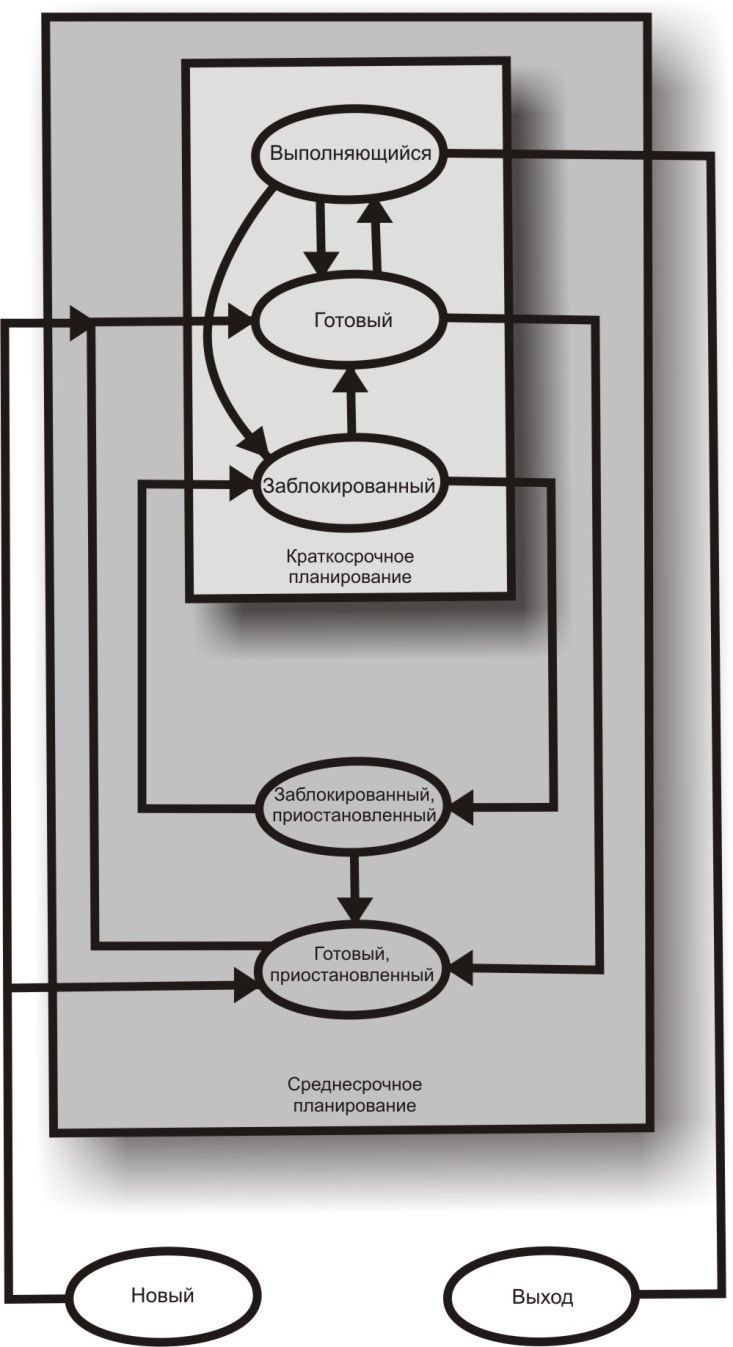


Рис 7.2. Уровни планирования

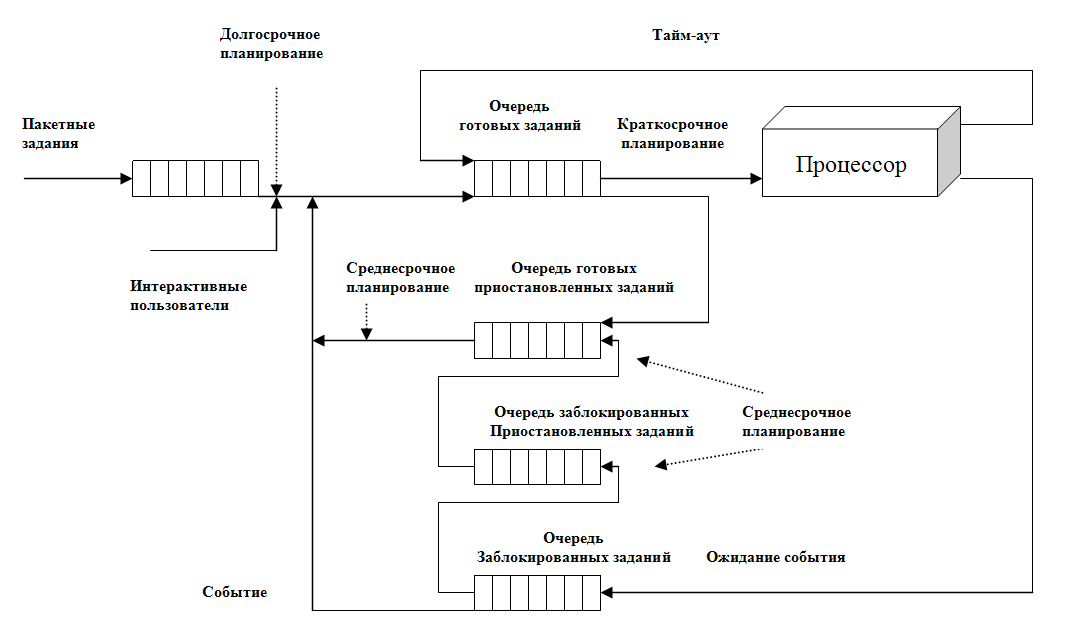


Рис. 7.3. Диаграмма планирования с участием очередей

**Долгосрочное планирование.**

Долгосрочное: планирование указывает, какие программы допускаются к выполнению системой, и тем самым определяет степень многозадачности. Будучи допущенным к выполнению, задание (или пользовательская программа) становится процессом, который добавляется в очередь для краткосрочного планирования. В некоторых системах вновь созданный процесс добавляется к очереди среднесрочного планировщика, будучи целиком сброшенным на диск.

В пакетных системах (или в пакетной части операционной системы общего назначения) новое задание направляется на диск и хранится в очереди пакетных заданий, а долгосрочный планировщик по возможности создает процессы для заданий из очереди. В такой ситуации планировщик должен принять решение, во-первых, о том, способна ли операционная система работать с дополнительными процессами, а во-вторых, о том, какое именно задание (или задания) следует превратить процесс (процессы).

Решение о том, когда следует создавать новый процесс, в общем определяется желаемым уровнем многозадачности. Чем больше процессов будет создано, тем меньший процент времени будет тратиться на выполнение каждого из них (поскольку в борьбе за одно и то же время конкурирует большое количество процессов). Таким образом, долгосрочный планировщик может ограничить степень многозадачности, с тем чтобы обеспечить удовлетворительный уровень обслуживания текущего множества процессов. Каждый раз при завершении задания планировщик решает,следует ли добавить в систему один или несколько новых процессов. Кроме того, долгосрочный планировщик может быть вызван в случае, когда относительное время простоя процессора превышает некоторый предопределенный порог.

Решение о том, какое из заданий должно быть добавлено в систему, может основываться на простейшем принципе «первым поступил −первым обслужен»; кроме того, для управления производительностью системы может использоваться и специальный инструментарий. Используемые в последнем случае критерии могут включать приоритет заданий, ожидаемое время выполнения и требования для работы устройств ввода-вывода. Например, если заранее доступна детальная информация о процессах, планировщик может пытаться поддерживать в системе смесь из процессов, ориентированных на вычисления и загружающих процессор, и процессов с высокой интерактивностью ввода-вывода и малой загрузкой процессора. Принимаемое решение может также зависеть от того, какие именно ресурсы ввода-вывода будут запрашиваться процессом.

В случае использования интерактивных программ в системах сразделением времени запрос на запуск процесса может генерироваться действиями пользователя по подключению к системе. Пользователи не просто вносятся в очередь в ожидании, когда система обработает их запрос на подключение. Вместо этого операционная система принимает всех зарегистрированных пользователей до насыщения системы (пороговое значение которого определяется заранее). После достижения состояния насыщения на все запросы на вход в систему будет получено сообщение о заполненности системы и временном прекращении доступа к ней с предложением повторить операцию входа попозже.

**Среднесрочное планирование.**

Среднесрочное планирование является частью системы свопинга. Обычно решение о загрузке процесса в память принимается в зависимости от степени многозадачности; кроме того, в системе с отсутствием виртуальной памяти среднесрочное планирование также тесно связано с вопросами управления памятью. Таким образом, решение о загрузке процесса в память должно учитывать требования к памяти выгружаемого процесса.

**Краткосрочное планирование.**

Рассматривая частоту работы планировщика, можно сказать, что долгосрочное планирование выполняется сравнительно редко, среднесрочное — несколько чаще. Краткосрочный же планировщик, известный также как диспетчер (dispatcher), работает чаще всего, определяя, какой именно процесс будет выполняться следующим.

Краткосрочный планировщик вызывается при наступлении события, которое может приостановить текущий процесс или предоставить возможность прекратить выполнение данного процесса в пользу другого. Вот некоторые примеры таких событий:

- прерывание таймера;

- прерывания ввода-вывода;

- вызовы операционной системы;

- сигналы.

**Алгоритмы плинирования**

**Критерии краткосрочного планирования**

Основная цель краткосрочного планирования состоит в распределении прицессорного времени таким образом, чтобы оптимизировать один или несколькоаспектов поведения системы. Вообще говоря, имеется множество критериев оценки различных стратегий планирования.

Наиболее распространенные критерии могут быть классифицированы в двух плоскостях. Во-первых, мы можем разделить их на пользовательские и системные. Пользовательские критерии связаны с поведением системы по отношению к отдельному пользователю или процессу. В качестве примера можно привести время отклика в интерактивной системе. Время отклика представляет собой интервал между передачей запроса и началом ответа на него. Его пользователь ощущает непосредственно, и, само собой,продолжительность интервала очень интересует его

Системные критерии ориентированы на эффективность и полноту использования процессора. В качестве примера можно привести пропускную способность, которая представляет собой скорость завершения процессов. Это, безусловно, эффективная мера производительности системы, которая должна быть максимизирована. Однако она в большей степени ориентирована на производительность системы, а не на обслуживание пользователя, так что и удовлетворять она будет системного администратора, а не пользователей системы.

В то время как пользовательские критерии важны почти для всех систем, системные критерии для однопользовательских систем не так значимы. В этом случае, пожалуй, достижение высокой эффективности использования процессора или высокая производительность не так существенны, как скорость ответа системы приложению пользователя.

Еще один способ разделения критериев — на те, которые связаны с производительностью, и те, которые с производительностью непосредственно не связаны. Ориентированные на производительность критерии выражаются числовыми значениями и обычно достаточно легко измеримы — примерами их могут служить время отклика и пропускная способность. Критерии, не связанные с производительностью непосредственно, либо качественны по своей природе, либо трудно поддаются измерениям и анализу. Примером такого критерия служит предсказуемость. Желательно, чтобы предоставляемые пользователю сервисы в разное время имели одни и те же характеристики, не зависящие от других задач, выполняемых в настоящее время системой. До некоторой степени этот критерий является измеримым − путём вычисления отклонений как функции от загрузки системы. Однако провести такие измерения оказывается вовсе не просто.

В табл. 7.2приведены ключевые критерии планирования. Все они взаимозави­симы, и достичь оптимального результата по каждому из них одновременно невоз­можно. Например, обеспечение удовлетворительного отклика может потребовать применения алгоритма с высокой частотой переключения процессов, что повысит накладные расходы и, соответственно, снизит пропускную способность системы. Следовательно, разработка стратегии планирования представляет собой поиск ком­промисса среди противоречивых требований; относительный вес каждого из крите­риев определяется природой и предназначением разрабатываемой системы.

В большинстве интерактивных операционных систем с одним пользователем или с разделяемым временем критичным требованием является время отклика.

Таблица 7.2. Критерии планирования

|  |  |
| --- | --- |
|  | **Пользовательские, связанные с производительностью** |
| **Время оборота** | Интервал времени между подачей процесса и его завершением. Включает время выполнения, а также время, затраченное на ожидание ресурсов, в том числе и процессора. Критерий вполне применим для пакетных заданий. |
| **Время отклика** | В интерактивных процессах это время, истекшее между подачей запроса и началом получения ответа на него. Зачастую процесс может начать вывод информации пользователю, ещё не окончив полной обработки запроса, так что описанный критерий − наиболее подходящий с точки зрения пользователя. Стратегия планирования должна пытаться сократить время получения ответа при максимализации количества интерактивных пользователей, время отклика для которых не выходит за заданные пределы. |
| **Предельный срок** | При указании предельного срока завершения процесса планирование должно подчинить ему все прочие цели максимализации количества процессов, завершающихся в срок. |
|  | **Пользовательские, иные** |
| **Предсказуемость** | Данное задание должно выполняться примерно за одно и то же количе­ство времени и с одной и той же стоимостью, независимо от загрузки системы. Большие вариации времени исполнения или времени отклика дезориентируют пользователей. Это явление может сигнализировать о больших колебаниях загрузки или о необходимости дополнительной настройки системы для устранения нестабильности ее работы. |
|  | **Системные, связанные с производительностью** |
| **Пропускная способность** | Стратегия планирования должна пытаться максимизировать количество процессов, завершающихся за единицу времени, что является мерой количества выполненной системой работы. Очевидно, что эта величина зависит от средней продолжительности процесса; однако на нее влияет и используемая стратегия планирования. |
| **Использование процессора** | Этот показатель представляет собой процент времени, в течение кото­рого процессор оказывается занят. Для дорогих совместно используе­мых систем этот критерий достаточно важен; в однопользовательских же и некоторых других системах (типа систем реального времени) этот критерий менее важен по сравнению с рядом других. |
|  | **Системные, иные** |
| **Беспристрастность** | При отсутствии дополнительных указаний от пользователя или системы все процессы должны рассматриваться как равнозначные и ни один процесс не должен подвергнуться голоданию. |
| **Использование приоритетов** | Если процессам назначены приоритеты, стратегия планирования должна отдавать предпочтение процессам с более высоким приоритетом. |
| **Баланс ресурсов** | Стратегия планирования должна поддерживать занятость системных ресурсов. Предпочтение должно быть отдано процессу, который недостаточно использует важные ресурсы. Этот критерий включает использование долгосрочного и среднесрочного планирования. |

**Использование приоритетов**

Во многих системах каждому процессу присвоен некоторый приоритет, и планировщик всегда должен среди процессов выбирать тот, у которого приоритет наибольший. На рис. 7.3 показано использование приоритетов. Вместо одной очереди готовых к исполнению процессов у нас имеется их множество, упорядоченное по убыванию приоритета: RQ0, RQ1, ..., RQn, т.е.

*Приоритет*[RQ*i*]>*Приоритет*[RQ*j*] при *i<j*

При выборе процесса планировщик начинает с очереди процессов с наи­высшим приоритетом (RQ0). Если в очереди имеются один или несколько про­цессов, процесс для работы выбирается с использованием некоторой стратегии планирования. Если очередь RQ0 пуста, рассматривается очередь RQ1 и т.д.

Одна из основных проблем в такой чисто приоритетной схеме планирования состоит в том, что процессы с низким приоритетом могут оказаться в состоянии голодания. Это будет происходить при постоянном поступлении новых готовых к выполнению процессов с высоким приоритетом. Если такое поведение нежелательно, приоритет процесса может снижаться при его выполнении.

**Альтернативные стратегии планирования**

В табл. 7.3 представлена некоторая информация о различных стратегиях планирования. Функция планирования определяет, какой из готовых к выполнению процессов будет выбран следующим для выполнения. Функция может быть основана на приоритете, требованиях к ресурсам или характеристиках выполнения процессов. В этом случае имеют значение три величины:

w − время, затраченное к этому моменту системой (ожидание и выполнение);

e − время, затраченное к этому моменту на выполнение;

s − общее время обслуживания, требующееся процессу, включая e (обычно эта величина оценивается или задаётся пользователем).

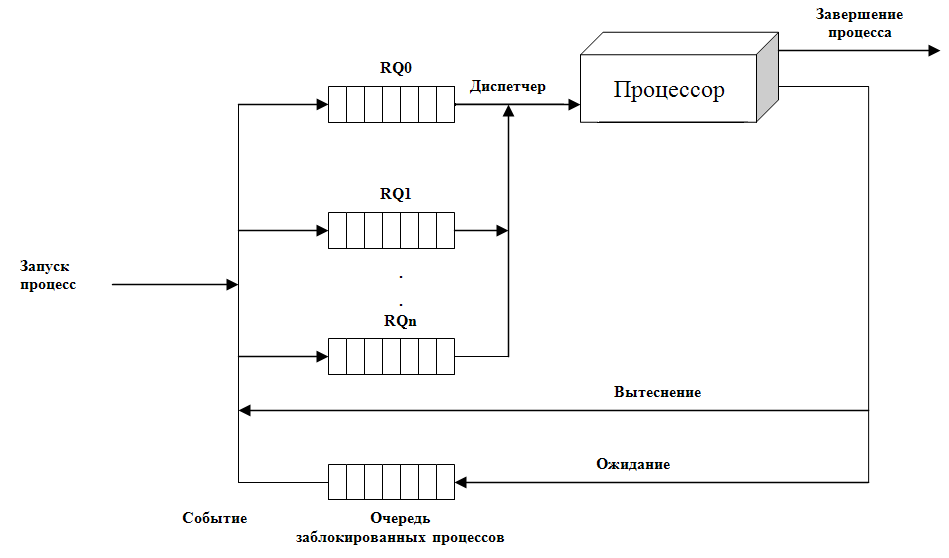


Рис 7.4. Планирование с учетом приоритетов

Например, выбор функции max[w] определяет стратегию "первым поступил - первым обслужен" (first-come-first-served — FCFS).

**Режим решения** определяет, в какие моменты времени выполняется функ-ция выбора. Режимы решения подразделяются на две основные категории.

**Невытесняющие**. В этом случае находящийся в состоянии выполнения процесс продолжает выполнение до тех пор, пока он не завершится или пока не окажется в заблокированном состоянии ожидания завершения операции ввода-вывода или запроса некоторого системного сервиса.

**Вытесняющие**. Выполняющийся в настоящий момент процесс может быть прерван и переведен операционной системой в состояние готовности к выполнению. Решение о вытеснении может приниматься при запуске нового процесса по прерыванию, которое переводит заблокированный процесс в состояние готовности к выполнению, или периодически — на основе прерываний таймера.

Вытесняющие стратегии приводят к повышенным накладным расходам по сравнению с невытесняющими, но при этом обеспечивают лучший уровень обслуживания всего множества процессов, поскольку предотвращают монопольное использование процессора в течение продолжительного времени одним из процессов. Кроме того, использование эффективных механизмов переключения про­цессов (по возможности реализованное аппаратно) и большой объем основной памяти (для хранения в ней как можно большего количества процессов) позволяющих поддерживать относительно небольшую стоимость вытеснения.

**Первым поступил − первым обслужен**

Простейшая стратегия планирования "первым поступил — первым обслу­жен" (first-come-first-served − FCFS) известна также как схема "первым при­шел— первым вышел", или схема строгой очередности. Как только процесс становится готовым к выполнению, он присоединяется к очереди готовых про­цессов. При прекращении выполнения текущего процесса для выполнения выбирается процесс, который находился в очереди дольше других.

Для однопроцессорных систем FCFS — не самая подходящая стратегия, но она часто комбинируется с использованием приоритетов. В этом случае плани­ровщик поддерживает ряд очередей, по одной для каждого уровня приоритета, и работает с процессами в каждой очереди в соответствии со стратегией FCFS.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Голодание** | Отсутствует | Отсутствует | Возможно | Возможно | Отсутствует | Возможно |
| **Влияние на процессы** | Плохо сказывается на коротких процессах и процессах с интенсивным вводом-выводом | Беспристрастна | Плохо сказывается на длинных процессах | Плохо сказываются на длинных процессах | Хороший баланс | Может привести к предпочтению процессов с интенсивным вводом-выводом |
| **Накладные расходы** | Минимальны | Минимальны | Могут быть высокими | Могут быть высокими | Могут быть высокими | Могут быть высокими |
| **Время отклика** | Может быть большим, в особенности при больших отклонениях во времени исполнения процесса | Обеспечивает хорошее время отклика для коротких процессов | Обеспечивает хорошее время отклика для коротких процессов | Обеспечивает хорошее время отклика | Обеспечивает хорошее время отклика | Не важно |
| **Пропускная способность** | Не важна | Может быть низкой при малом времени | Высокая | Высокая | Высокая | Не важна |
| **Режим решения** | Невытесняющий | Вытесняющий | Невытесняющий | Вытесняющий (по решению) | Невытесняющий | Вытесняющий (по времени |
| **Функция выбора** | max[w] | const | min[s] | Min[s-e] | min([w+s]/s) | См.текст |
|  | **FCFS** | **Круговая** | **SPN** | **SRT** | **HRRN** | **Со снижением приоритета** |

**Круговое планирование**

Очевидный путь повышения эффективности работы с короткими процесса­ми в схеме FCFS — использование вытеснения на основе таймера. Простейшая стратегия, основанная на этой идее, — стратегия кругового (карусельного) пла­нирования (roundrobin — RR). Таймер генерирует прерывания через определенные интервалы времени. При каждом прерывании исполняющийся в настоящий момент процесс помещается в очередь готовых к выполнению процессов, и начинает выполняться очередной процесс, выбираемый в соответствии со стратегией FCFS. Эта методика известна также как **квантование времени** (timeslicing), поскольку перед тем как оказаться вытесненным, каждый процесс получает квант Iвремени для выполнения.

При круговом планировании принципиальным становится вопрос о продолжи­тельности кванта времени. При малом кванте времени короткие процессы будут относительно быстро проходить через систему, но при этом возрастают накладные расходы, связанные с обработкой прерывания и выполнением функций планирования. Следовательно, очень коротких квантов времени следует избегать. Одно из полезных правил в этом случае звучит так: квант времени должен быть немного больше, чем время, требующееся для типичного полного обслуживания. Если квант оказывается меньшего размера, большинство процессов потребует как минимум два кванта времени.

**Выбор самого короткого процесса**

Еще один путь к снижению перекоса в пользу длинных процессов — ис­пользование стратегии выбора самого короткого процесса (shortestprocessnext — SPN). Это невытесняющая стратегия, при которой для выполнения выбирается процесс с наименьшим ожидаемым временем исполнения.

Основная трудность в применении стратегии SPN состоит в том, что для ее осуществления необходима по меньшей мере оценка времени выполнения требующегося каждому процессу.

Основной риск при использовании стратегии SPN заключается в возможном голодании длинных процессов при стабильной работе коротких процессов. Кроме того, хотя SPN снижает перекос в пользу длинных процессов, его применение нежелательно в системах с разделением времени или системах обработки транзакций из-за отсутствия вытеснения

**Наименьшее остающееся время**

Стратегия наименьшего остающегося времени (shortestremainingtime — SRT) представляет собой вытесняющую версию стратегии SPN. В этом случае планировщик выбирает процесс с наименьшим ожидаемым временем до окончания процесса. При присоединении нового процесса к очереди готовых к исполнению процессов может оказаться, что его оставшееся время в действительности меньше, чем оставшееся время выполняемого в настоящий момент процесса. Планировщик, соответственно, может применить вытеснение при готовности нового процесса. Как и при использовании стратегии SPN, планировщик для корректной работы функции выбора должен оценивать время выполнения процесса; в этом случае также имеется риск голодания длинных процессов.

В случае использования стратегии SRT нет таких больших перекосов в пользу длинных процессов, как при использовании стратегии FCFS; в отличие от стратегии RR, здесь не генерируются дополнительные прерывания, что снижает накладные расходы. Тем не менее в этом случае происходит увеличение накладных расходов из-за необходимости фиксировать и записывать время выполнения процессов. В связи с тем что короткие задания немедленно получают преимущество перед выполняющимися длинными заданиями, стратегия SRT существенно выигрывает у стратегии SPN во времени оборота.

**Справедливое планирование**

Все рассмотренные алгоритмы планирования рассматривают множество готовых к выполнению процессов как единый пул, из которого выбирается очередной процесс для выполнения. Этот пул может быть разделен по степени приоритета процессов, но в противном случае он остается гомогенным.

Однако в многопользовательских системах при организации приложений или заданий отдельных пользователей как множества процессов (или потоков) у них имеется структура, не распознаваемая традиционными планировщиками. С точки зрения пользователя, важно не то, как будет выполняться отдельный процесс, а то, как будет выполняться множество процессов, составляющих единое приложение. Таким образом, было бы неплохо, если бы планирование осуществлялось с учетом наличия таких множеств процессов. Данный подход в целом известен как справедливое (fair-share) планирование. Эта же концепция может быть распространена на группы пользователей, даже если каждый из пользователей представлен единственным процессом. Например, в системе с разделением: времени мы можем рассматривать всех пользователей данного отдела как членов одной группы. Планировщик принимает решения с учетом необходимости предоставить каждой группе пользователей по возможности одинаковый сервис. Таким образом, если в системе находится много пользователей из одного отдела, то изменение времени отклика должно в первую очередь коснуться именно пользователей этого отдела, не затрагивая прочих пользователей.

Термин *справедливое планирование* указывает на философию, лежащую в основе такого планирования. Каждому пользователю назначен определенный вес, который определяет долю использования системных ресурсов данным пользователем. В частности, каждый пользователь использует процессор. Данная схема работает более или менее линейно, так что. если вес пользователя А в два раза превышает вес пользователя В, то в течение достаточно длительного промежутка времени пользователь А должен выполнить в два раза большую работу, чем пользователь В. Цель справедливого планирования состоит в отслеживании использования ресурсов и предоставлении меньшего количества ресурсов тому пользователю, который уже получил лишнее, и большего количества — тому, чья доля оказалась меньше справедливой.

Был предложен ряд алгоритмов справедливого планирования. В этом разделе мы рассмотрим схему планирования, реализованную в ряде систем UNIX. Эта схема известна как справедливый планировщик (fair-sharescheduler — FSS). FSS при принятии решения рассматривает историю выполнения связанной группы процессов вместе с индивидуальными историями выполнения каждого процесса. Система разделяет пользовательское сообщество на множество групп со справедливым планированием и распределяет процессорное время между ними. Так, если у нас имеется четыре группы, каждая из них получит по 25% процессорного времени. В результате каждая группа обеспечивается виртуальной системой, работающей, соответственно, медленнее, чем система в целом.

Планирование осуществляется исходя из приоритетов с учетом приоритета процесса, недавнего использования им процессора и недавнего использования процессора группой, к которой он принадлежит. Чем больше числовое значение приоритета, тем ниже сам приоритет. Для процесса j из группы *k*применимы следующие формулы:

**

**

**

где  — мера загруженности процессора процессом j на интервале i;

 — мера загруженности процессора группой *k*на интервале i;

 — приоритет процесса *j*в начале интервала *i*(меньшее значениесоответствует большему приоритету);

 — базовый приоритет процесса *j;*

*Wk—* вес, назначенный группе *k*( и ).

Каждому процессу назначается базовый приоритет. Приоритет процесса снижается по мере использования им процессора, так же как и по мере использования процессора группой в целом. В случае использования процессора группой среднее значение нормализуется делением на вес группы. Чем больший вес назначен группе, тем меньше использование ею процессора влияет на приоритет.

**Традиционное планирование в Unix**

В этом разделе рассмотривается традиционное планирование UNIX, используемое как в SVR3, так и в 4.3 BSDUNIX. Эти системы в первую очередь предназначены для работы в интерактивной среде с разделением времени. Алгоритм планирования разработан таким образом, чтобы обеспечить приемлемое время отклика для интерактивных пользователей, в то же время гарантируя отсутствие голодания низкоприоритетных заданий. Хотя описываемый алгоритм и был заменен в более современных версиях UNIX, его изучение как представителя практически используемых алгоритмов с разделением времени не лишено основания. Схема планирования SVR4 соответствует требованиям реального времени.

Традиционный планировщик UNIX использует многоуровневый возврат с применением кругового планирования в пределах очередей каждого приоритета, а также односекундное вытеснение. Таким образом, если текущий процесс не блокируется или не завершается в пределах одной секунды, он вытесняется. Приоритет основан на типе процесса и истории выполнения. Применяются следующие формулы:

**

**

где  — мера использования процессора процессом j на протяжении интервала j;

 — приоритет процесса *j*в начале интервала *i*(меньшее значениесоответствует большему приоритету);

 — базовый приоритет процесса *j;*

— указываемый пользовательский коэффициент.

Приоритет каждого процесса пересчитывается один раз в секунду, в момент принятия решения о том, какой процесс будет выполняться следующим. Назначение базового приоритета состоит в разделении процессов на фиксированные группы уровней приоритетов. Значения компонентов CPU и nice ограничены требованием того, чтобы процесс не мог выйти из назначенной ему на основании базового приоритета группы. Эти группы используются для оптимизации доступа к блочным устройствам (например, к диску) и обеспечения быстрого отклика операционной системы на системные вызовы. Имеются следующие группы приоритетов (приведены в порядке снижения приоритетов):

- программа свопинга;

- управление блочными устройствами ввода-вывода;

- управление файлами;

- управление символьными устройствами ввода-вывода;

- пользовательские процессы.

Такая иерархия должна обеспечить наиболее эффективное использование устройств ввода-вывода. В группе пользовательских процессов использование истории исполнения приводит к применению штрафных санкций к процессам, ориентированным на вычисления, что также должно способствовать повышению эффективности системы. В сочетании с круговой схемой с вытеснением данная стратегия удовлетворяет требованиям общего назначения с разделением.

***Тема 12. Управление вводом-выводом и файлами***

Внешние устройства, по отношению к которым выполняются операции ввода-вывода могут быть объединены в три группы:

1) работающие с пользователем – связывают пользователя с вычислительной машиной (видеотерминалы, манипулятор, мышь, принтер);

2) работающие с компьютером (дисковые устройства, датчики, контроллеры), которые служат для связи с электронным оборудованием;

3) устройство коммуникации служит для связи с удаленным устройством (модемы, драйверы цифровых линий и т.д.);

Устройства ввода-вывода классифицируются по следующим характеристикам.

1. Скорость передачи данных.
2. Применение. Каждое действие, поддерживаемое устройством, оказывает влияние на программное обеспечение и стратегии операционной системы. Для размещения файла на диске требуется управление файловой системы. Для организации виртуальной памяти нужны программные средства, реализующие функции подкачки. С видеотерминалом может работать обычный пользователь и администратор.
3. Сложность управления.
4. Единицы передачи данных. Данные могут передаваться блоками или как поток байтов.
5. Представление данных. Разные устройства используют различные схемы кодирования данных, включая контроль четности и помехоустойчивые коды.
6. Условия ошибок. Природа ошибок, способ сообщения о них последствия, возможные ответы существующего отличия при переходе от одного устройства к другому.

**Организация функций ввода-вывода**

Существует три способа ввода-вывода.

1. Программируемый ввод-вывод. Процессор посылает необходимые команды контроллеру ввода-вывода и после этого процессор находится в состоянии ожидания завершения операции ввода-вывода;
2. Ввод-вывод, управляемый прерываниями. Процессор посылает нужные команды контроллеру ввода-вывода и продолжает выполнение следующих команд. Выполнение процесса прерывается контроллером ввода-вывода, когда устройство готово к передаче данных. После посылки команды, процессор выполнит или текущий процесс, если не требуется ожидание данных от устройств ввода-вывода, или процессор переключается на выполнение другого процесса;
3. Прямой доступ к памяти. Модуль прямого доступа к памяти управляет обменом данных между основной памятью и контроллером ввода-вывода. Процессор посылает запрос на передачу блока данных модулю прямого доступа к памяти. После завершения обмена модуль ПДП может инициировать прерывание либо процессор проверяет бит готовности;

**Развитие функций ввода-вывода**

Этапы развития функциональности устройств ввода-вывода.

1. Процессор непосредственно управляет периферийными устройствами.
2. К устройству добавляется контроллер или модуль ввода-вывода. Процессор использует программируемый ввод-вывод без прерываний. На этом этапе процессор отдаляется от конкретных деталей внешних устройств.
3. Та же конфигурация, что и на втором этапе, но еще используются прерывания. Повышение производительности достигается за счет того, что не требуется ожидать завершение операции ввода-вывода, а появление запроса на прерывание инициирует операции обмена.
4. Модуль ввода-вывода получает возможность работы с памятью через прямой доступ к памяти (ПДП или DMA). На этом этапе блоки данных перемещаются без использования процессора за исключением начальных этапов.
5. Модуль ввода-вывода совершенствуется и становится отдельным процессором, обладающим специальной системой команд для ввода-вывода. Центральный процессор задает процессору ввода-вывода задание выполнить программу ввода-вывода, находящуюся в основной памяти. Процессор ввода-вывода производит выборку и выполнение соответствующих команд без участия центрального процессора. Это позволяет центральному процессору определить последовательность выполняемых функций ввода-вывода и быть прерванным только на завершающей фазе ввода данных.
6. Модуль ввода-вывода обладает своей локальной памятью и, по сути, является отдельным компьютером. Здесь управление различными устройствами при выполнении операций ввода-вывода осуществляется при минимальном вмешательстве центрального процессора. Такая архитектура используется для управления связью с интерактивными терминалами. Процессор ввода-вывода выполняет большинство задач, связанных с управлением терминалами.

Этот путь развития устройств ввода-вывода ориентирован на то, чтобы вмешательство центрального процессора в функции ввода-вывода становилось минимальным. Изменилась концепция устройства ввода-вывода, в результате чего устройство ввода-вывода способно само выполнять программу.

**Управление ОС устройствами ввода-вывода**

При проектировании средств ввода-вывода имеется 2 цели: эффективность и универсальность.

Эффективность важна в том смысле, что операции ввода-вывода часто является основным тормозом производительности всей системы, т.к. они работают медленнее, чем центральный процессор. Один из подходов решения – многозадачность.

Универсальность служит для снижения вероятности возникновения ошибок и упрощения работы с устройствами ввода-вывода. Желательно иметь возможность одинакового управления различными устройствами, как со стороны пользовательских процессов, так и со стороны ОС. Реально достичь большей универсальности не удается, но применение модульного подхода при разработке функций ввода-вывода возможно. ОС представляет определенный уровень иерархии, и каждый уровень представляет некоторое подмножество функций необходимых операционной системе. Некоторые части ОС должны взаимодействовать непосредственно с аппаратным оборудованием компьютера, где продолжительность события в проявлении нескольких микросекунд, а другие части ОС работают с пользователем, который вводит команды один раз в несколько секунд. Исходя из такого подхода, рассматривается несколько моделей организации ввода-вывода.

**Модели организации ввода-вывода**

Пользовательские процессы

Пользовательские процессы

Пользовательские процессы

Логический порт вывода

Управление каталогами

Файловая система

Аппаратное обеспечение

Физическая организация

Устройство ввода-вывода

Планированиеи управление

Аппаратное обеспечение

Планирование и управление

Устройство ввода-вывода

Архитектура связи

Устройство ввода-вывода

Планирование и управление

Аппаратное обеспечение

Модель для локального Коммуникационный Файловая

устройства порт система

Рис. 8.1.

Функции ОС распределяются по уровням. Выполнение примитивных функций передается более низкому уровню. Более высокий уровень не знает деталей выполняющихся задач на низком уровне. С другой стороны, любой уровень обеспечивает обслуживание верхнего уровня.

Логический ввод-вывод. На этом уровне ведется обращение к устройствам как к логическим ресурсам и не уделяется внимание деталям фактического управления устройства. Логический модуль ввода-вывода является посредником между пользовательскими процессами и устройством, предоставляя процессам набор высокоуровневых функций.

Устройство ввода-вывода. Запрошенные операции и данные преобразуются в соответствующие последовательности инструкций ввода-вывода, команды управления каналом, команды контроллера. Здесь может применять буферизация.

Планирование и управление. На этом уровне производится реальная организация очередей и планирование операций ввода-вывода, а также управление выполнением операций. Осуществляется работа с прерываниями, получение и передача информации о состоянии устройства. Программное обеспечение на этом уровне непосредственно взаимодействует с аппаратурой устройства.

Управление каталогами. На этом уровне происходит преобразование символьных имен файлов в идентификаторы, указывающие на файл непосредственно или косвенно с использованием файлового дескриптора или индексной таблицы. На этом уровне организованы пользовательские операции работы с каталогами файлов: добавление, удаление, реорганизация.

Файловая система. Работает с логической структурой файлов и операциями над ними: открытие, закрытие, чтение, запись. Управление правами доступа также определяется на этом уровне.

Физическая организация. Логические ссылки на файлы и записи должны быть конвертированы в физические адреса конкретного внешнего запоминающего устройства с учетом физической структуры дорожек и секторов внешнего запоминающего устройства. На этом же уровне происходит общее управление распределением пространства внешней памяти и буферов основной памяти.

***Тема 13. Аппаратно-программные особенности современных*** ***процессоров, ориентированные на поддержку многозадачности***

**Сегментация памяти**

**Сегментация памяти в процессорах 8086**

Под **сегментом** понимается блок смежных ячеек памяти в адресном пространстве 1Mb, с максимальным размером в 64Kb и начальным (базовым) адресом, находящимся на 16-байтной границе (**параграфе**). Для обращения к памяти необходимо определить базу сегмента и 16-битное расстояние от базы, называемое **смещением** .

Схема сегментации памяти для этого случая приведена на рис. 9.1.

Смещение

База сегмента

Память

00000H

FFFFFH

64Кбайт

15 0

15 0

16 байт

Данные

Рис. 9.1. Сегментация памяти в процессоре 8086

СЕГМЕНТ : СМЕЩЕНИЕ логический (виртуальный) адрес

Физический адрес ФА = СЕГМЕНТ \*16 + СМЕЩЕНИЕ (\*)

Чтобы упростить обращения к памяти, за каждой командой закрепляется сегментный регистр по умолчанию, т.е. команды адресуются через ***CS:IP***, стек через ***SS:SP***, данные через ***DS:[смещение]*** и ***ES:[смещение].***

Преобразование пары СЕГМЕНТ : СМЕЩЕНИЕ в физический производится в соответствии с выше приведенным выражением и, в результате, получаем 20-битный физический адрес. Переход от логического адреса в физический однозначен, а из физического адреса в логический неоднозначен, т.е. каждому физическому адресу может быть сопоставлено в соответствие 4Kb логических адресов.

При таком подходе есть неприятный момент, называемый ***заворачиванием адреса,*** возникающий, когда рассчитываемое значение физического адреса больше 1Mb.

Сегментация памяти процессора 8086 имеете особенности, усложняющие разработку многозадачных систем:

1. Сегменты памяти имеют всего два атрибута:

- начальный адрес (на границе параграфа);

- максимальный размер (64 Kb).

Никаких аппаратных средств контроля правильности использования сегментов нет.

1. Размещение сегментов в памяти произвольно, т.е. они могут частично или полностью перекрываться или не иметь общих частей.
2. Программа может обратиться к любому сегменту для выполнения операций чтения/записи или для выборки команды. Программа может обратиться по любому физическому адресу, а для защиты памяти от несанкционированного доступа требуются внешние схемы.
3. Нет препятствий для обращения к даже несуществующей физической памяти. Если программа выдаёт адрес несуществующей памяти, то результат работы зависит от аппаратных особенностей конкретной вычислительной машины.

**Сегментация памяти в процессорах архитектуры IA-32**

Сегменты описываются отдельными информационными структурами, называемыми ***дескрипторами.***

Основное отличие данной модели сегментации памяти заключается в том, что пользовательская программа не может свободно обращаться по любому адресу в пространстве памяти. Программа, в зависимости от уровня привилегий, не может обращаться к сегменту до тех пор, пока он не описан для данной программы.

Определение сегментов в многозадачном режиме производится на системном уровне, причем определяется следующая информация для каждого сегмента:

- базовый адрес;

- размер сегмента;

- целевое использование;

- уровень привилегий;

- другие параметры.

Базовый адрес может быть произвольным, т.е. выравнивание не требуется на границу параграфа или страницы. Размер сегмента может задаваться от 1 байта до 4 Гбайт. Это значит, что с помощью 32-битного индекса можно пройти всё адресное пространство, не модифицируя сегментный регистр.

**Дескриптор сегмента**

Каждый сегмент характеризуется 8-байтная структурой данных, которая называется ***дескриптор сегмента ( SegmentDescriptor).***

Права доступа

AR

Базовый адрес GDXU Предел PDPLS Тип AБазовый

31-24 19-16 адрес 23-16

Базовый адрес Предел

15-0 15-0

63 47 39 32

31 23 15 7 0 0

Рис. 9.2. Формат дескриптора сегмента

Поля дескриптора имеют следующие названия.

***AR*** *- Access Rights*

***DPL*** *- Descriptor Privilege Level*

***D*** *- Default size*

***U*** *- User*

***A*** *- Accessed*

***G*** *- Granularity*

***P*** *- Present*

***S*** *- System(segment)*

***AVL -*** *Available*

Число дескрипторов в системе практически неограниченно. Если какую-то область адресного пространства не описать дескриптором, соответствующий диапазон адресов оказывается недоступным и процессор блокирует доступ к этим адресам.

***Поля***

***Базовый адрес* -** это 32-х битное поле. Занимает 2,3,4,7 байты дескриптора и определяет начальный адрес сегмента в линейном адресном пространстве 4 Gb. Этот адрес формирует процессор при нулевом смещении.

***Предел*** - 0,1 байты и 4 бита- граница сегмента(20 бит). Он равен размеру сегмента в байтах минус 1. Предел задает последнюю адресуемую единицу в сегменте. 20-битное поле предела позволяет определить размер в 1 Mb элементов.

***Бит G*** (*гранулярность)-* определяет единицы измерения элементов памяти:

*- G=0,* то поле предела измеряется в байтах и соответственно имеет размер 1 Mb;

*- G=1,* то единицей измерения является страница, каждая из которых имеет размер 4Kb, следовательно максимальный размер 1 M\*4Kb = 4 Gb.

***Бит A-*** *бит доступа.* Автоматически устанавливается в 1, когда производится обращение к памяти, описанной данным дескриптором. Этот бит можно использовать для того, чтобы определить те сегменты, к которым долго не было обращения.

***Бит D****-* размер по умолчанию. Он используется для совместимости с процессором 286.

***Бит U-*** пользовательский. Его называют AVL. Предназначен для использования при системном программировании.

***Бит AR*** *- байт прав доступа.*

P DPL 1 1 C R A

P DPL 1 0 ED W A

P DPL 1 x x x x

S Тип

7 6 5 4 3 2 1 0

Сегмент

кода

Сегмент

данных

Системный

объект

Рис. 9.3. Формат байта прав доступа

***Байт прав доступа*** определяет тип сегмента и те возможности, которые предоставляются программе при работе с этим сегментом.

Для всех типов дескрипторов в байте доступа имеется *бит P (присутствие*):

*P=1,* когда описываемый дескриптором сегмент находится в физической памяти. В системе с виртуальной организацией памяти часть сегментов располагается на вешней дисковой памяти, бит *P = 0* , и если программа обращается к сегменту, у которого в дескрипторе бит *Р = 0*, то возникает особый случай неприсутствия сегмента. В таком случае ОС должна найти свободную область в ОП, скопировать с диска содержимое сегмента в эту область, загрузить в дескриптор новый базовый адрес сегмента, установить *P = 1* и осуществить повторный запуск команды, вызвавшей особый случай. Процесс обращения к не присутствующему сегменту с последующей передачами сегмента между основной памятью и диском называется ***свопингом (подкачкой-swapping)***. Если свободного места для размещения сегмента в ОП нет, то ОС должна выгрузить некоторые сегменты из ОП на дескрипторы этих сегментов и *Р = 0.*

***DPL*** *– уровень привилегий дескриптора сегмента* (2 бита). Определяет уровень привилегий, ассоциируемый с той областью памяти, которую описывает дескриптор. Поэтому его следовало бы назвать уровень привилегий сегмента. Наивысший уровень привилегий: *DPL =00* (ноль), а наименьший *– DPL =11* (три). Поле *DPL* - составная часть механизма защиты в процессоре.

***Бит S****(системный, бит сегмента).* В дескрипторах сегментов памяти этот бит всегда равен 1. А в дескрипторах системных объектов *S=0.*

*Трех битное поле типа* определяет целевое использование сегмента, задавая допустимые в сегменте операции.

Тип:

**000** *–* сегмент данных, разрешено только считывание;

**001***–* сегмент данных, разрешено считывание и запись;

**010***–* сегмент стека, разрешено только считывание;

**011***–* сегмент стека, разрешено считывание и запись;

**100***–* сегмент кода, разрешено только выполнение;

**101***–* сегмент кода, разрешено выполнение и считывание;

**110***–* подчиненный сегмент кода, разрешено только выполнение;

**111***–* подчиненный сегмент кода, разрешено выполнение и считывание.

*Отдельныебиты:*

***R -*** *Read;*

***ED-*** *Expand Down;*

***W -*** *Write;*

***C -*** *Conforming.*

***R****–* показывает возможность считывания кода как данных с помощью префикса замены сегмента. Если *R = 1*, то можно описать другой дескриптор и адресацию можно производить не через *CS*, а через *DS* и *ES.*

***С*-*бит*** *подчинения*. Если *С = 1*, то соответствующий сегмент кода лишается защиты по уровню привилегий и при обращении к этому сегменту (фактически запуск программы, описывающей этот сегмент) его уровень привилегий переводится на уровень привилегий вызывающей программы. Таким образом он позволяет создать системные библиотеки доступные пользовательским программам.

Для сегмента данных ***W*-*бит***показывает возможность записи в сегмент данных или, по-другому, возможность изменения его содержимого. Если этот бит равен нулю, то из сегмента можно читать данные, т.е. описывать виртуальное пространство как постоянное запоминающее устройство, похоже на ПЗУ.

***Бит ED****-бит расширения вниз*. С помощью этого бита можно определить сегмент стека (ED=1) и сегмент данных (ED=0). Этот бит управляет интерпретацией поля предела. Размер сегмента памяти равен 8 Кб.

С

Т

Е

К

База

+0FFFH

База

+1FFFH

База

База

База

+1FFFH

Д

а

н

н

ы

е

56

Кбайт

Недоступны

Сегмент

стека

Сегмент

данных

Рис. 9.4. Интерпретация поля предела для сегментов данных и стека

Таким образом определяется ситуация, при которой максимальный адрес находится вверху, а минимальный адрес (0) *–* внизу. Минимальный адрес задается для сегмента стека полем предела, а значение максимального адреса зависит от значения *бита* ***D****(размер по умолчанию*). Если *D=0*, то максимальный адрес: база + FFFFh, т.е. 64K-1. А если бит *D=1*, то максимальный адрес равен: база + constFFFFFFFFh.

***Бит X****–* резервный.

Такая организация памяти позволяет сделать достаточно надежную систему защиты (изолированность кодов и данных).

**Дескрипторные таблицы**

***Дескрипторные таблицы*** – области памяти, предназначенные для хранения 8-байтных дескрипторов. Порядок расположения дескрипторов в таблице не имеет значения, а максимальное число дескрипторов в таблице равно 8192, т.е. максимальный размер дескрипторной таблицы – 64 Kb.

Процессор предусматривает использование трех типов дескрипторных таблиц.

1. *глобальные дескрипторные таблицы* ***GlobalDescriptionTable (GDT****)*
2. *дескрипторнаятаблицапрерываний****Interrupt Description Table (IDT****)*
3. *локальныедескрипторныетаблицы****Local Description Table (LDT****)*

Предел Базовый адр.

Предел Базовый адр.

…

**Дескриптор 1**

…

…

Предел Базовый адр.

**Дескриптор N**

**Дескриптор 0**

**Дескриптор M**

**Дескриптор 1**

Дескриптор 0

**Дескриптор K**

Дескриптор 1

Дескриптор 0

47

47

47

0

0

0

0

Регистр *GDTR*

Регистр *IDTR*

Дескриптор *LDTR*

+

+

+

*GDT*

*IDT*

*LDT*

15LDTR

Память

Рис. 9.5. Выбор дескриптора из таблицы

Главной общесистемной таблицей дескрипторов является глобальная дескрипторная таблица GDT. Все программы, выполняющиеся в системе, могут использовать эту таблицу дескрипторов для обращения к сегментам памяти. Место размещения *GDT* в памяти хранится в регистре глобальной дескрипторной таблицы *GDTR*. *GDTR –* 48-битный. В нем хранится 32-битный базовый адрес и 16-битное поле предела. Предел имеет размер в байтах и если необходимо, например, в таблице хранить *N* дескрипторов, то поле предела будет равно *8·N-1*.

Дескрипторная таблица прерываний (*IDT*) является тоже общесистемной. Она содержит специальные системные объекты или дескрипторы специальных системных объектов, которые называются *шлюзами*, определяющими точки входа подпрограмм обработки прерываний и особых случаев, т. е. эта таблица заменяет таблицу векторов прерываний процессора 8086. Месторасположение этой таблицы определено в регистре *IDTR*.

В многозадачной системе для каждой задачи можно создать свою дескрипторную таблицу (*LDT*), в которой будут определены сегменты, доступные только конкретной задаче. Такую *LDT*определяет 16-битный регистр *LDTR*, что позволяет хранить дескрипторы, описывающие *LDT,* в *GDT*.

С регистром LDTR связан *теневой* регистр. При загрузке селектора в регистр *LDTR* в этот теневой регистр загружается дескриптор из *LDT*, что позволяет ускорит обращение к дескрипторам *LDT.* Это также позволяет рассматривать *LDT* как обычные сегменты памяти и применять к ним операции своппинга.

Система команд процессора имеет всего шесть команд (привилегированных), которые могут выполнять операции с регистрами дескрипторных таблиц. Эти команды позволяют записывать данные в эти регистры и считывать данные из этих регистров.

***LGDT*** *mem48 L-Load-загрузить*

***LIDT*** *mem48 S-Save-сохранить*

***SGDT*** *mem48*

***SIDT*** *mem48*

***LLDT*** *reg16/mem16*

***SLDT*** *reg16/mem16*

Операнд *mem48* – это 48-битная структура памяти, первые 16 бит которой содержат предел, а следующие 32 бита - линейный базовый адрес.

Для *LLDT*и*SLDT* (команд работы с LDT) аргументом должен быть правильный селектор *LDT.*

**Селекторы сегментов в защищенном режиме**

Для выбора дескриптора из дескрипторной таблицы используются данные, которые загружаются в сегментные регистры. В защищенном режиме размер сегментного регистра 16 бит, однако его содержание интерпретируется следующим образом.

**Индекс TI RPL**

15

3

1

0

2

Рис. 9.6. Селектор сегмента в защищенном режиме

***RPL -*** *Requested Privilege Level.*

***TI -*** *Table Indicator.*

2-битное поле ***RPL***(поле уровня запрашиваемых привилегий) в выборе дескриптора не участвует, а используется для контроля привилегий в механизме защиты.

Бит ***TI*** *– идентификатор таблицы* – показывает из какой дескрипторной таблицы следует выбрать дескриптор. Если *TI=0*, то из *GDT*, а если *TI=1*, то из *LDT.*

***Последовательность действий по преобразованию логического адреса в линейный адрес.***

1. Взять селектор из сегментного регистра.
2. Если бит *TI=0*, т.е. бит указывает на *GDT*, то из регистра *GDTR* выбрать адрес дескрипторной таблицы и перейти к шагу 4.
3. Если бит *TI=1* , т.е. бит указывает на *LDT*, то:

3.1. Взять селектор сегмента *LDT* из регистра *LDTR*;

3.2. Выделить в селекторе поле индекса, умножить этот индекс на 8 и т.о. получить смещение внутри *GDT*, которое указывает на расположение дескриптора *LDTR*;

3.3. Прибавить полученное значение к адресу начала *GDT* из регистра *GDTR*;

3.4. Считать из памяти адресуемый дескриптор;

3.5. Выделить из этого дескриптора базовый адрес сегмента, содержащего таблицу LDT, полученный адрес является адресом дескрипторной (локальной) таблицы;

1. Выделить значение из поля индекса селектора, умножить его на 8 и прибавить к адресу дескрипторной таблицы и считать дескриптор по этому адресу;
2. Выделить из дескриптора базовый адрес сегмента;
3. Прибавить значение смещения к базовому адресу сегмента, в результате получим линейный адрес
4. Осуществить обращение к памяти в соответствие с полученным линейным адресом

Если отмеченные действия производить каждый раз при преобразовании адреса, то время преобразования адреса было бы значительно больше, чем при работе процессора в реальном режиме.

Index TI

GDT

Права База Предел

LDT

Права База Предел

0

31

63

0

63

0

31

0

15

0

47

0

15 3 0

IndexLDTR

Эффективный адрес

Базовый адрес сегмента

0

31

Линейный адрес

Линейный базовый адрес Предел

Рис. 9.7. Схема преобразования адреса

Для ускорения таких преобразования адреса в процессоре имеются специальные ***теневые регистры***, связанные с сегментными регистрами и *LDTR*. Раз

мер этих регистров 64 бита и при загрузке нового значения либо в сегментный регистр, либо в *LDTR,* в соответствующий теневой регистр помещается в выбранный дескриптор из дескрипторной таблицы. Применение теневых регистров позволяет время преобразования адреса при работе в защищенном режиме сделать почти равным времени преобразования адреса при работе процессора в реальном реме.

Некоторые особенности загрузки селекторов в сегментные регистры.

До загрузки селектора в сегментный регистр и соответствующего выбора дескриптора процессор выполняет несколько проверок. Ряд этих проверок связан с контролем уровня привилегий в механизме защиты, а другие предотвращают загрузку бессмысленных селекторов.

* + - 1. Процессор проверяет, чтобы поле индекса селектора находилось в пределах таблицы, определяемой полем TI, поэтому пределы дескрипторных таблиц хранятся вместе с их базовыми адресами.
      2. При загрузке селекторов в сегментные регистры данных тип дескриптора должен разрешать считывание из сегмента. Только выполняемые дескрипторы для этих регистров не допускаются.
      3. При загрузке регистра ***SS*** (сегментный регистр стека) в сегменте должны быть разрешены операции чтения и записи.
      4. При загрузке регистра ***СS*** сегмент должен быть обязательно выполняемым.
      5. Если эти проверки кончились успешно, то процессор анализирует бит присутствия в дескрипторе, и только если этот бит равен 1, разрешается загрузка сегментного регистра и загрузка соответствующего дескриптора в теневой регистр. В противном случае, процессор инициализирует прерывания по особым случаям и загрузка селектора не производится. Когда выбираемый селектором дескриптор находится вне предела дескрипторной таблицы или дескриптор имеет неверный тип, процессор формирует нарушение общей защиты (прерывание 13h). Если селектор отмечен как не присутствующий, то генерируется нарушение неприсутствия (прерывание 11h). Если ошибка связана с регистром SS, то генерируется прерывание 12h. При генерировании нарушений, ошибочный селектор включается в стек, и процедуры обработки особых случаев могут выяснить причину нарушения. В отличие от процессора 8086 в защищенном режиме по селектору не возможно узнать какое адресное пространство определяют адресные регистры. Селектор определяет только номер дескриптора, описывающего адресное пространство.

Для упрощения анализа дескриптора и с целью допущения инициирования заведомо запрещенных действий имеется несколько команд, через которые можно получить информацию о дескрипторах.

***LAR reg16/32, reg16/mem16*** *Load Access Rights Загрузкаправдоступа*

***LSL reg16/32, reg16/mem16*** *Load Segment Limit Загрузкапредела*

***VERR reg16/mem16*** *Verity for Read Проверканасчитывание*

***VERW reg16/mem16*** *Verity for Write Проверканазапись*

При выполнении этих команд процессор учитывает уровень привилегий текущих программ и тех сегментов, которые проверяются с помощью этих команд. При нарушении прав привилегий информация не возвращается.

**Локальные дескрипторные таблицы**

Локальная дескрипторная таблица представляет собой массив восьмибайтных дескрипторов.

В любой момент времени процессор работает лишь с одной LDT, а при переключении задач изменяется и активная LDT, т.к. дескрипторы, описывающие сегменты, где хранятся локальные таблицы, заносятся в GDT, то для локализации или определения используемой в текущий момент времени LDT, необходим только селектор. И для его хранения в процессоре используется 16-битный регистр LDTR. Поле предела занимает 20 бит, поэтому можно создать таблицу более 64 килобайт. Но на практике этого не требуется. В дескрипторе также задействован бит присутствия, и процессор не разрешает загрузить в регистр LDTR селектор, не присутствующий в LDT, генерируется особый случай неприсутствия.

Для начала работы с LDT в регистр LDTR помещается селектор выбираемой таблицы. При попытке загрузить в селектор недопустимое значение процессор формирует общее нарушение защиты. Допускается загрузка в LDTR пустого селектора. Нулевой селектор означает, что таблица LDT не используется.

К LDI обращение на прямую не допускается.

Дескриптор таблицы LDTR подобен общей структуре дескриптора.

Базовый адрес 0 0 0 0 Предел PDPL 0 Тип Базовый

31-24 19-16 0 0 1 0 адрес 23-16

Базовый адрес Предел

15-0 15-0

63 55 47 S 39 32

31 23 15 7 0 0

Рис. 9.8. Формат дескриптора таблицы LDT

**Особенности сегментации**

Разрешается создание сегментов, в которых допускаются операции только считывания, только исполнения, сегменты считывания - записи, сегменты исполнения - считывания. Не предусмотрено создания сегментов для всех трех функций. Однако путем перекрытия сегментов или введения дескрипторов с альтернативными наименованиями позволяют определять участки памяти, по отношению к которым можно выполнять все эти 3 действия одновременно. CPU обеспечивает мощные средства для сегментации памяти (защита, фиксация недопустимых операций и т.д.)

**Страничная организация памяти**

Внутреннее устройство управления памяти в процессоре, наряду с обязательной сегментацией, реализует еще один уровень косвенности в формировании физического адреса памяти. Это страничная организация памяти.

Основной причиной применения страничного преобразования является возможность реализации виртуальной памяти, которая позволяет программисту использовать объем памяти больше чем размер физической ОП.

Страничное преобразование было реализовано и до процессоров INTEL, но здесь оно реализовано как внутреннее устройство. В связи с этим появился ряд преимуществ:

- сокращается загрузка внешней шины;

- гарантируется правильность взаимодействия процессора и страничного устройства управления памятью;

- стандартизируются средства поддержки ОС виртуальной памяти;

- допускается программное разрешение и запрещение страничного преобразования адреса;

- снижается общая стоимость системы.

Если при сегментации базовым объектом памяти является сегмент практически любого размера, то при страничной организации базовым объектом памяти является блок с фиксированным размером 4 kb, который называется *страницей*. При линейном адресном пространстве 4 Gb получается 1 MB страниц. Реальная физическая память меньше 4 Gb, следовательно, при выполнении программы в любой момент времени только часть страниц будет присутствовать в ОП. Такой подход в организации памяти называется *виртуальной памятью.* Страницы, которые не поместились в ОП, располагаются на внешней памяти.

Фиксированный размер всех страниц позволяет загрузить любую виртуальную страницу в любую физическую страницу. Это и объясняет применение страничного преобразования в случае виртуальной памяти.

Понятие виртуальной памяти с заменой страниц по требованию подчёркивает тот факт, что прикладная программа не касается процесса страничного преобразования памяти и может использовать всё доступное адресное пространство.

Процессор автоматически формирует особый случай неприсутствия, если обращение идёт к странице, отсутствующей в физической памяти. При возникновении этого особого случая ОС автоматически загружает затребованную страницу из внешней памяти.

В общем случае сегменты не являются кратными размеру страницы. Но рекомендуется выравнивать небольшие сегменты так, чтобы они полностью находились внутри одной страницы.

В процессе страничного преобразования старшие 20 бит 32-битного линейного адреса замещаются другим 20-битным значением, задающим адрес физической страницы.

Для такого преобразования используются таблицы страниц. В процессорах реализуется двухэтапное преобразование линейного адреса в физический. Можно реализовать и одноэтапное преобразование, но тогда для одной таблицы страниц нужно 4 Мб. Поэтому целесообразнее использование двухэтапного преобразования.

**. ..**

**PDE**

**. . .**

. . .

**PTE**

**. . .**

Страничный

кадр

Байт

PDBR

Каталог Таблица Смещение

Каталог

страниц

31 0

1023

31 … 0

1023

Физическая

память

CR3

31 22 21 12 11 0

19 0

1023

0

0

0

Таблицы

страниц

Рис. 9.9. Двухэтапное преобразование линейного адреса в физический

**PDE** – Page Directory Entry.

**PTE** – Page Table Entry.

**PDBR** – Page Directory Base Register.

Основой страничного преобразования является регистр CR3, который содержит 20-битный физический адрес каталога страниц в текущей задаче, и называется регистром базового адреса каталога страниц (PDBR). Это единственный внутренний регистр процессора, который содержит физический адрес памяти. Младшие 12 бит считаются нулевыми, т.е. каталог выровнен на границу страницы.

Предполагается, что каталог страниц всегда находится в памяти и не участвует в свопинге.

Каталог страниц содержит 1024 32-битных дескриптора, называемых элементами каталога страниц. Каждый из этих элементов адресует подчинённую таблицу страниц второго уровня. В свою очередь, каждая из этих страниц содержит 1024 32-битных дескриптора элементов таблицы страниц, а каждый из этих элементов адресует страничный кадр в физической памяти.

Преобразование линейного адреса в физический состоит из следующих действий.

1. Старшие 10 бит линейного адреса дополняются двумя младшими нулями и служат индексом каталога страниц, выбирая один из элементов в каталоге. Выбранный элемент каталога страниц определяет 20-ти битный адрес таблицы страниц.

2. Средние 10 бит линейного адреса дополняются двумя младшими нулями и индексируют таблицу страниц, выбирая из нее PTE. Этот элемент содержит 20-ти битный базовый адрес страничного кадра в физической памяти.

3. Базовый адрес страничного кадра объединяется с младшими 12-ю битами линейного адреса. В результате получается 32-х битный физический адрес памяти, по которому и производится обращение.

При обращении к элементам каталога и таблицы страниц, производится проверка защиты и присутствия страниц в памяти.

Структура элемента каталога страницы и таблицы страниц одинакова и имеет вид.

PPUR

Адрес страничного кадра Дост. 0 0 DACW / /

D TS W W

31 12 11 9 0

P

Рис. 9.10. Формат элемента таблицы страниц

**A** – Accessed

**D** – Dirty

**PWT** – Page Write Through

**PCD** – PageCacheDisable

В каждом элементе страниц хранится адрес страничного кадра. В этом поле находится физический базовый адрес страницы, младшие 12 бит адреса считаются нулевыми (выравниваются на границу страницы), биты 9, 10, 11 процессор не использует, и они могут использоваться при разработке операционных систем.

Бит присутствия Р показывает, отображается ли адрес страничного кадра на страницу в физической памяти. Если Р = 1, то страница находится в памяти.

Бит пользовательский или системный U/S определяет права доступа, т. е. уровни привилегий.

Бит обращения А и бит D содержат информацию об использовании страницы. Бит D сообщает об обращении к таблице для записи, а бит А сообщает об обращении для чтения или записи к странице или таблице страниц второго уровня. За исключением бита D в элементе каталога страниц, эти биты устанавливаются в единицу аппаратно, но самостоятельно не сбрасываются, т. е. сбрасываются программно. Бит А используется для нахождения наиболее часто используемых страниц, а бит D используется при перезагрузке страниц. Анализируя бит D можно определить изменена ли страница в ОП.

Биты R/W (чтения и записи) применяются в механизме защиты применительно к страницам.

Биты PCD, PWT (запрещение кэширования страниц и сквозная запись) применяются для управления кэшированием на уровне страниц.

Механизм сегментации не касается страничного преобразования адреса. В процессе сегментации участвуют только логические и линейные адреса, и нет ни одного обращения к памяти по физическому адресу. Когда действует страничное преобразование, линейный и физический адреса обозначают совершенно различные объекты. Адреса физической памяти содержатся только в регистре CR3 и 20-битных полях базовых адресов элементов каталога страниц и таблиц страниц.

При разработке ОС, когда реализуется функции создания дескрипторов, речь идет о линейных адресах, а при создании таблиц для страничного преобразования уже преобразуются и физические адреса.

**Страничный дескриптор**

Страничный дескриптор (рис. 9.10) содержит следующую информацию.

Р – вид присутствия. Он показывает, находится ли данная страница в физической памяти. Если Р установлен в 0, то при попытке использовать данный элемент возникает особый случай страничного нарушения и операционная система, обрабатывая это страничное нарушение, должна выполнить действие по размещению физической операционной памяти выбранной страницы. После того, как страница загружена, в него загружается адрес, бит Р устанавливается в 1. После этого повторяется команда, приведшая к страничному нарушению.

r/w – чтения и записи. Показывает какие действия можно выполнить над страницей;

u/s – бит прав доступа. Если при сегментации используется и привилегии, то для страничного только или на системном уровне или на пользовательском.

D и A используются для определения использования страницы. А = 1, если было обращение к странице. D = 1, если осуществилась запись в станицу, т.е. в странице изменились данные.

Эти биты используются при организации виртуальной памяти. D показывает, что если страница не изменилась, то нет смысла ее перезаписывать.

Механизм сегментации не касается страничного преобразования адреса. В процессе сегментации участвуют только логические и линейные адреса и нет обращения к памяти по физическому адресу. Когда включено страничное преобразование, линейные и физические адреса обозначают совершенно различные объекты. Хотя в принципе можно так создать каталог страниц и таблицу страниц, когда линейные адреса будут отражаться точно в такие же адреса – такая ситуация называется тождественным отображением. Прикладные программы имеют дело только с логическими адресами. При разработке операционной системы, когда реализовываются функции создания дескрипторов сегментов, речь идет о линейных адресах. А при создании таблиц для страничного преобразования уже требуются знания физических адресов. Каталог страниц и таблицы страниц размещаются в удобном месте и требуют таких дескрипторов сегментов, которые имеют их как данные с разрешенными операциями чтения и записи.

Современные процессоры поддерживают режим расширенного физического адреса до 64 бит. А общий размер памяти до 64 Гбайт.

**PAE** – PhysicalAddressExtensions

В этом режиме блок страничной операции оперирует 64 битными элементами. Причем сохраняя режим поддержки и 32 битных элементов. В этом режиме поддерживается размер страницы 2 ГБ и схема преобразования адреса имеет несколько другую структуру.

**Разрешение и запрещение страничного преобразования**

Перед переходом в режим страничного преобразования адресов, создаются каталог страниц и таблицы страниц, которые размещаются в удобном месте ОП. Таблицы занимают блоки линейного адресного пространства и для их описания применяются дескрипторы, которые описывают эти блоки, с разрешенными операциями чтения и записи. Страничное преобразование включается последним из основных средств процессора. Для этого устанавливается в 1 старший бит в регистре CR0. И после этого, со следующей команды начинается страничное преобразование. Для того, чтобы не произошел переход после включения страничного преобразования в другой диапазон адресов, необходимо выделить область памяти, где осуществляется прямое отображение.

Для того, чтобы преобразование произошло корректно, рекомендуется выполнить следующие 3 действия:

1. Запретить аппаратные прерывания, включая и немаскируемые, если это возможно.

2. Разрешать страничное преобразование только из страницы с тождественным отображением.

3. После команды mov, которая установит в 1 бит PG, следует очистить очередь команд устройства предвыборки.

pushfd ; сохранить состояние флага if

cli ; сброс if

moveax, dir\_base ;загрузить в бит cr3 базовый адрес каталога страниц

movcr3, eax

moveax, cr0 ; установить в 1 бит PG, разрешить страничное преобразование

btseax, 31 ; установлен в 1 31-й бит

movcr0, eax

jmpnext

next: nop ; очищается очередь предвыборки команд

popfd ; восстановить if

Переход от страничного преобразования также должен идти с предосторожностями. При страничном преобразовании активно используется ассоциативный буфер страничного преобразования, в котором осуществляется кэширование элементов таблиц страниц и каталога страниц.

**.3 Сравнение сегментной организации памяти и страничной**

**организации памяти**

Достоинства сегментации.

1. Возможность реализации виртуальной памяти

2. Разработка надежных и живучих операционных систем на базе механизма защиты по привилегиям

3. Автоматическое обнаружение и обработка программных ошибок, вызванных неверными указателями или нарушением предела.

4. Сегментация имеет гибкую схему реализации(можно создать 1 сегмент кода и 1 сегмент данных, которые включают в себя все адресное пространство). При этом, программа видит только свое адресное пространство, а с сегментами работает только ОС. Защита осуществляется только по задачам, и можно создать полностью сегментную систему, когда каждому большому объему данных или кода выделяется отдельный сегмент. В результате получается надежная среда выполнения программ, но ухудшается быстродействие, по причине частой перезагрузки сегментных регистров.

*Основным недостатком сегментации* является необходимость выполнения служебных функций при загрузке селекторов в системные регистры. В ходе выполнения этой операции необходимо загружать в теневой регистр 8-байтный дескриптор сегмента из соответствующей дескрипторной таблицы, а также должна производится установка бита А в 1, в соответствующем дескрипторе дескрипторной таблицы. А также к недостаткам сегментации относится фрагментация памяти.

*Основным достоинством страничной организации памяти* по сравнению с сегментацией является фиксированный размер страницы. Это позволяет:

– решить проблему организации памяти в общем (удобно отганизовывать виртуальную память);

– согласовать и обеспечить высокую скорость передачи данных между внешней памятью и ОП.

Страничная организация памяти не видна прикладному программисту.

*Недостатки страничной организации памяти*.

1. Имеет место эффект *внутренней фрагментации*, когда сегменты памяти имеют размеры, не кратные размеру страниц;

2. Увеличение времени преобразования адреса, т.к. обращение к каталогу страниц и таблиц страниц требуется для каждой операции обращения к памяти (хотя компенсируется аппаратной организацией).

**Организация защиты при работе процессора в защищенном режиме**

При работе в защищенном режиме допускается одновременное выполнение нескольких прикладных программ, но они изолированы друг от друга таким образом, что ошибки одной программы не влияли на другие программы и операционную систему.

Когда программа совершила неожиданное обращение к недопустимому для неё пространству памяти, механизм защиты блокирует обращение и сообщает о его возникновении. Однако следует понимать, что в однопроцессорной системе можно реализовать только виртуальную многозадачность, т.е. выполнение только одной задачи в один момент времени, но имеется возможность быстрого переключения между задачами. Для целей защиты предусматривается как минимум два режима работы:

– системный режим (режим супервизора);

– пользовательский режим;

В режиме супервизора, в котором работает операционная система, доступны все ресурсы системы. При работе в пользовательском режиме накладывается ряд ограничений по выполнению некоторых команд процессора, влияющих на общие системные ресурсы (некоторые команды ввода-вывода, управление системными ресурсами, прерываниями и т.п.).

В процессорах Intel обеспечивается аппаратная поддержка защиты по 4 уровням привилегий.

Средства защиты должны предотвращать:

– неразрешенное взаимодействие пользователей друг с другом;

– несанкционированный доступ к данным;

– повреждение программ и данных из-за ошибок в других программах;

– преднамеренные попытки разрушить целостность системы;

– случайное искажение данных.

Механизм защиты процессоров Intel делится на две части:

- управление памятью;

- защита по привилегиям.

Схемы управления памятью обнаруживают большинство программных ошибок (например, формирование неверных адресов, нахождение индекса за пределами массива, искажения стека и т.д.), а защита по привилегиям позволяет выявить более тонкие ошибки и преднамеренные попытки нарушить функционирование системы.

При работе в защищенном режиме процессор постоянно контролирует достаточно ли привилегированна текущая программа для того, чтобы:

- выполнять некоторые команды;

- обращаться к данным других программ;

- передавать управление внешнему коду по отношению к самой программе с помощью команд дальней передачи управления:***farcall*** или ***farjmp***.

**Привилегированные команды**

К ним относятся те, которые модифицируют состояние флажка *if*, изменяют сегментацию, или изменяют сам механизм защиты, а также команды ввода/вывода.

*Команды, воздействующие на механизм сегментации и защиты*, могут выполнятся только на нулевом уровне привилегий:

***hlt*** ;остановка процессора;

***clts***  ;сброс флага переключенной задачи;

***lgdt, lidt, lldt*** ;загрузка регистров дескрипторной таблицы;

***ltr***  ;загрузка регистра задач;

***lmsw***  ;загрузка слова состояния машины;

К этой группе также относятся команды передачи данных в регистры управления и проверки.

Вторую группу образовывают команды, связанные с изменением флажка прерываний и команды, производящие ввод/вывод.

Эти команды могут быть выполнены программой, у которых уровень привилегий меньше либо равен уровню *IOPL (э*то поле, находящееся в регистре флажков). Их еще называют *IOPL-чувствительными командами.* Изменить значение этого бита флажка прерывания и поля *IOPL*, который находится в регистре флажков. Казалось бы, можно обходным путем с помощью команды занесения регистра флажков в стек и извлечения из стека, которые не относятся к привилегированным. Но процессор всё равно контролирует уровень привилегий программы, которые выполняет эти команды и, если обнаруживается, что попытку изменить указанные биты предпринимает программа, которая не имеет на это право (не относящихся к нулевому уровню), но процессор запрещает модификацию этих битов. Для того, чтобы программа могла выполнять IOPL*–*чувствительные команды надо, чтобы CPL<=IOPL.

**Защита доступа к данным**

Для работы любой программы требуется адресное пространство данных и стека. Процессор не разрешает обращаться к данным, которые более привилегированны, чем выполняемая программа.

Основное правило защиты доступа к данным формулируется следующим образом: . Т.е. текущий уровень привилегий выполняемой программы должен быть меньше или равен уровню привилегий дескриптора сегмента данных, к которому идет обращение.

При выполнении команд обращения к данным процессор

1. проверяет привилегии при загрузке селектора в один из сегментных регистров данных (*DS, ES, FS, GS*). Если условие не выполняется, то селектор не загружается, и формируется ситуация нарушения общей защиты и изменение сегментного регистра не производится (блокировка).

2. После успешной загрузки селектора, при использовании его для фактического обращения к памяти, процессор контролирует, чтобы запрашиваемая операция чтения или записи для этого сегмента была разрешена.

При загрузке селектора в сегментный регистр стека правила защиты ужесточаются, а именно , т.е. запрещается использовать стек даже с меньшим уровнем привилегий. Если P=1, то выбираемый сегмент должен обязательно присутствовать в памяти.

- это правило при обращении к сегменту данных.

***RPL-*** *Requested Privilege Level (запрашиваемыйуровеньпривилегий)*

***DPL***- Descriptor Privilege Level

***EPL****- Effective Privilege Level (эффективныйуровеньпривилегий)*

***CPL****- Current Privilege Level*

В системе защиты используется поле RPL. Защита по принципу CPL<=DPL позволяет контролировать код и данные на различных уровнях привилегий. А для предотвращения использования ошибочных указателей, которые передают более привилегированным программ используется понятие эффективного уровня привилегий max(CPL,RPL) = EPL и с учетом этого должно выполняться условие EPL<= DPL. Если RPL<CPL, то поле RPL значения не имеет и RPL = 0. Если RPL = 0, то контроль идет по текущему уровню привилегий того сегмента, к которому идет обращение.

**.3 Защита сегмента кода**

Процессоры Intel запрещают передачу управления сегменту кода, находящемуся на другом уровне привилегий. Это самый сложный механизм защиты и самый важный. Помимо прочего он имеет несколько исключений. Ограничивая передачу управления в пределах одного кольца защиты, процессор предотвращает изменений уровней привилегий. Как известно, передачу управления в другой сегмент выполняют команды дальнего перехода, дальнего вызова процедуры и возврата из процедуры (***far call, farjmp и ret***соответственно). Адрес передачи управления задается с помощью 48-битного указателя селектор:смещение, который содержится либо в самой команде, либо берётся из памяти. При выполнении этих команд изменяется значение регистра **CS**и **EIP.**

С точки зрения процессора контроль межсегментной передачи управления заключается в проверке достоверности селектора, загружаемого в регистр *CS*. При загрузке селектора в регистр *CS* процессор выполняет следующие проверки.

1. Проверяет, что целевой дескриптор определяет сегмент кода, т.е. имеет атрибут выполняемого сегмента.

2. *CPL* должен быть равен *DPL* целевого сегмента*.*

3. Целевой сегмент кода должен быть отмечен присутствующим и новое значение регистра указателя команд должно находится в пределах нового сегмента кода.

Если какая-либо проверка даёт отрицательный результат, то формируется ошибка общей защиты (int ). При невыполнении этих проверок формируется нарушения общей защиты или исключение не присутствия.

**Определение текущего уровня привилегий**

*CPL* –это уровень привилегий выполняющегося кода. Он задается полем *RPL* селектора в регистре *CS*. Поэтому имеется возможность выполняемой программе определить уровень привилегий, на которых она выполняется.

**mov ax, cs** или **push cs**

**and ax, 03h pop ax**

**and ax, 03h**

Два способа передачи:

1. использование подчиненных сегментов;
2. использование специальных дескрипторов называется шлюзами вызова.

**Передача управления между уровнями привилегий**

Когда пользовательские программы взаимодействуют с операционной системой, возникает потребность передачи управления с низкого уровня привилегий на уровень привилегий операционной системы. Для таких передач есть два способа. Первый более простой и называется использование подчиненных сегментов. Второй более сложный – использование специальных дескрипторов, названных шлюзами вызовов.

**Подчиненные сегменты**

Сегмент кода определяется как подчиненный, если бит c в байте прав доступа дескриптора сегмента установлен в 1. при обращении к таким сегментам обычное правило защиты CPL = DPL не действует, действует тока правило, что CPL>= DPL, е.у. можно передавать управление на более высокий или текущий уровень привилегий. При передаче управления на подчиненный сегмент два младших бита регистра CS не изменяются. Таким образом, выполнение программы будет производится на том же уровне, на котором выполнялась вызывающая программа. Такой способ передачи управления используется при обращении к функциям операционной системы, которая не требует изменить состояние системы и не требует работы с внешними устройствами. Передача управления через подчиненные сегменты может осуществляться только во внутренние, более защищенные уровни привилегий.

**Шлюзы вызова**

Шлюзы вызова позволяют реализовать фактическое изменение уровня привилегий.

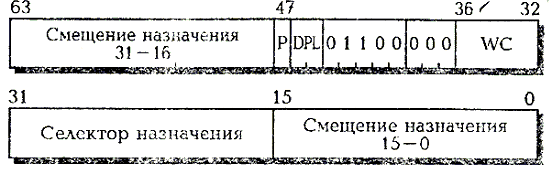


Рис. 9.11. Формат дескриптора шлюза вызова

Поле*WC(WordCount) - счетчик слов,* определяет число параметров, которые передаются в вызываемую программу; максимально можно передать в вызываемую программу через шлюз вызова 128 байт.

Шлюз вызова определяет точку входа программы.

Дескриптор шлюза вызова определяет полный указатель (селектор + смещение) точки входа в процедуру назначения, которой передается управление. Дескриптор шлюза вызова – это своеобразный интерфейсный слой между сегментами кода, находящимися на различных уровнях привилегий.

Шлюзы вызова определяют разрешенные точки входа в более привилегированный код и являются единственным средством смены уровня привилегии.

Дескрипторы шлюзов вызова не определяют никакого адресного пространства, поэтому у них нет полей базы и предела. По своей сути это даже не дескрипторы, но их размещают либо в глобальной дескрипторной таблице, либо, при необходимости, в локальных дескрипторных таблицах. Селекторы для выбора дескрипторов шлюзов вызова необходимо загружать только в сегментный регистр *CS* и ни в какие другие сегментные регистры.

Адресовать шлюз вызова можно только в команде межсегментного вызова ***far call,*** использование ***far jmp*** запрещено. Сама команда ***call*** должна адресовать шлюз вызова, а не сегмент кода назначения.

**CALL**

Сегмент

Смещение

Разрешено

Запрещено

Память

Дескриптор шлюза

Дескриптор кода

**…**

Сегмент кода с DPL=0

Смеще

ни

е

Рис. 9.12. Разрешенный и запрещенный вызовы более

привилегированного кода

Реализованный в Intel процессорах косвенный вызов привилегированных процедур имеет несколько преимуществ:

1. Привилегированный код сильно защищен, и вызывающие его программы не могут его разрушить. При этом предполагается, что сам код такой процедуры тщательно отлажен и не содержит ошибок.

2. Шлюзы вызова делают привилегированные процедуры невидимыми для программ на внешних уровнях привилегий.

3. Так как вызывающая программа прямо адресует только шлюз, реализуемые процедурой функции можно изменить или переместить их в адресном пространстве, не затрагивая интерфейс со шлюзом.

**Правила защиты при использовании шлюза вызова**

1. ;

2. ;

3. .Это правило предотвращает передачу на более низкий уровень привилегий;

4. .

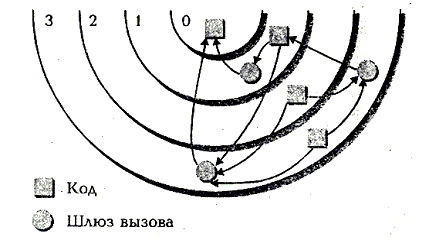


Рис. 9.13. Разрешенные варианты передачи управления через шлюз вызова

**Поддержка многозадачности в процессорах архитектуры IA-32**

Под задачей понимается программа или группа связанных программ, которая выполняется в многозадачном (мультипрограммном) режиме и её выполнение не должно подвергаться воздействиям извне (в смысле другими программами), и в свою очередь, она не должна оказывать влияния на действия, выполняемые другими программами. В процессорах Intel архитектуры задача определяется как совокупность кода и данных, которым назначен сегмент состояния задачи, т.е. сегмент состояния задачи является эквивалентным контекстной памяти, в которой хранится информация о задачи, когда она не выполняется и откуда она берется при повторном старте задачи. Сегмент состояния задачи – небольшой сегмент данных с разрешенными операциями чтения и записи, доступ к которым запрещен всем программам и к сегменту состояния задачи может обращаться только сам процессор.

Для управления многозадачностью в процессоре нет специальных команд. Вместо этого в некоторых случаях по-другому интерпретируются команды межсегментной передачи управления. Переключение задачи может быть вызвано командами межсегментной передачи управления (***farcall*** и ***farjump***). Помимо этого новая задача может активизироваться прерыванием или особым случаем. Когда реализуется одна из таких форм передачи управления, по типу адресуемого дескриптора процессор определяет что ему нужно сделать: выполнить обычную межсегментную передачу или переключить задачу.

Имеется два типа дескрипторов, относящихся к задачам:

1. Дескриптор сегмента состояния задачи.

2. Шлюз задачи.

При каждом переключении задачи процессор может перейти к другой локальной дескрипторной таблице.

Переключение задачи похоже на вызов процедуры, но при этом сохраняется намного больше информации. Сохраненная информация должна обеспечивать возобновление работы задачи с той же точки и в том же объеме что и когда задача была прервана.

Для поддержки многозадачности в процессоре имеются:

- регистр задачи;

- сегмент состояния задачи;

- дескриптор сегмента состояния задачи;

- дескриптор шлюза задачи.

Типичным примером многозадачной работы являются системы разделения времени. В ходе функционирования такой системы требуются средства, позволяющие приостановить выполнение задачи, временно сохранить ее состояние, восстановить состояние задачи, на которую осуществляется переключение и инициировать ее продолжение работы.

**Сегмент состояния задачи**

Сегмент состояния задачи представляет собой контекстную память, в которой сохраняется информация о задаче, когда она не выполняется и откуда берется информация при рестарте задачи.

База двоичной карты разрешения ввода-вывода

Системно-зависимая часть

31 15 0

1. T 64H
2. LDTR 60H

0 GS 5CH

0 FS 58H

0 DS 54H

0 SS 50H

0 CS 4CH

0 ES 48H

EDI 44H

ESI 40H

68 H

EBP 3CH

ESP 38H

EBX 34H

EDX 30H

ECX 2CH

EAX 28H

EFLAGS 24H

EIP 20H

CR3 1CH

0 SS2 18H

ESP2 14H

0 SS1 10H

ESP1 CH

0 SS0 8H

ESP0 4H

0 Обратная связь 0

Рис. 9.14. Формат 32-битного сегмента TSS

Часть сегмента состояния задачи размером в 104 байта является обязательной. Эту часть процесса сохраняет и восстанавливает автоматически. Как и любой другой сегмент TSS определяется дескриптором, который называется дескриптором сегмента состояния задачи, который может находится тока в глобальной дескрипторной таблице.

Сохранение регистров флагов позволяет восстановить условия выполнения предыдущей команды. Бит *T –* бит ловушки. Он используется при отладке.

В поле обратной связи хранится селектор TSS задачи, которая выполнялась перед текущей. С его помощью может быть организована цепь вложенных задач подобно цепочке вложенных подпрограмм.

Когда процессор начинает выполнение новой задачи, он считывает всю информацию из первых 104 байтов TSS (минимальный размер сегмента состояния задачи). После этого выполняется команда, которая адресуется регистрами *CS* и *EIP.* При переключении задач между задачами никакой информации не передается. Этим исключается искажения задач и обеспечивается возможность переключения задач в любой момент времени.

База 31-24 0 0 0 U Предел PDPL 0 1 0 B 1 Базовый

19-16 адрес(23-16)

Базовый адрес Предел

15-0 15-0

63 47 S 39 32

31 0

Рис. 9.15. Формат дескриптора сегмента состояния задачи

Сохранение состояния выполняемой задачи требует сохранения содержимого всех регистров процессора, некоторых переменных и адреса команды, которая должна выполняться после рестарта задачи. Это информация по задаче называется её *контекстом*, а действия процессора по сохранению состояния задачи и рестарта другой задачи называются *переключением контекста.*

В ЭВМ выделяется область памяти, доступная только ОС в которую записывается контекст задачи. Требуется чтобы всем задачам, работающим в системе, выделялись такие области памяти, и они постоянно находились в ОП машины. Это обеспечит быстрое переключение задач, но требует определенных затрат ОП (Поле дескриптора TSS стандартное).

**Дескриптор сегмента TSS**

Дескриптор сегмента состояния задачи определяет сегмент состояния задачи. Поле *DPL* этого дескриптора показывает, какие программы по уровню привилегий могут обращаться к задаче, определенной данным дескриптором TSS, т.е. функции этого поля аналогичны функциям *DPL* в шлюзе вызова.

Поле типа показывает активна задача или нет. Задачи не являются реентерабельными (т.е. не допускается вызов самой себя или повторный вызов), поэтому бит *В* имеет существенное значение. Он показывает занята задача или нет, если бит в 1 – это значит что задача выполняется, т. е. занята.

Поле предела имеет размер не менее 67h, это минимальный допустимый размер сегмента состояния задачи. При попытке переключения на задачу, дескриптор которой имеет меньший предел, чем 67h, то в дескрипторе возникает особый случай, но больший предел может быть(когда применяется база двоичной карты ввода/вывода).

Обращение к дескриптору сегмента состояния инициализирует переключение задачи. Во многих ОС, поле DPL этих дескрипторов должно содержать 0, поэтому переключение задач могут производить высоко привилегированные программы. Программам не предоставляется возможность считать или модифицировать этот дескриптор. Модификацию можно произвести, применяя дополнительные дескриптор данных, описывающий ту же область памяти, дескрипторы TSS могут располагаться только в GDT.

Загрузка дескриптора в сегментный регистр приводит к особому случаю. Инициализируется особый случай, когда производится обращение к дескриптору сегмента состояния задачи с установленным битом *TI*. Не предусмотрено сохранение значения сопроцессора, не все задачи использует сопроцессор и его сохранение нужно производить только когда следующая задача начинает изменять его состояние.

**Сегмент состояния задач TSS**

Большая часть этого сегмента отведена для хранения внутренних регистров процессора и для ряда сегментных регистров, таких, как DTRCR3 и другие.

В этом сегменте также хранятся 3 указателя на стеки, для 0, 1 и 2-го уровня привилегий. Для каждой задачи допускается образование своего каталога страниц и LDT. В поле обратной связи сохраняется селектор состояния той задачи, которая выполнялась перед текущей. С помощью этого поля можно организовать цепочку вложенности задач.

Бит T – это бит ловушки, который применяется при отладке программы. Кроме того, сохраняется в TSS 16-битное слово, которое определяет смещение начала двоичной карты размещений разрешения ввода-вывода. Эта карта помещается в памяти произвольно, но обязательно вблизи сегмента TSS. Эта карта используется при контроле привилегий для команд ввода-вывода. Следовательно, она может занимать 8 Кб памяти. Если бит, для соответствующего порта установлен в 1, то попытка задачи обратиться к этому порту приводит к формированию особого случая нарушения общей защиты, т. е. таким образом можно разделить порты между отдельными задачами. При переключении задач между ними, никакой информации не передается, поэтому они максимально друг от друга, кроме того, при переключении задач сохраняется весь контекст старой задачи, но перепись указателей на стеки не производится в целях экономии времени, потому что эти указатели считаются постоянными на всё время существования задачи.

**События, которые могут вызвать переключение задачи**

Используются термины "выходящая (старая) задача" и "входящая (новая) задача".

Переключение задач может быть вызвано четырьмя событиями.

1. Выходящая задача выполняет команду ***farcall***или ***farjmp***, и селектор выбирает шлюз задачи.

2. Выходящая задача выполняет команду ***farcall*** или ***farjmp***, и селектор выбирает дескриптор *TSS*.

3. Выходящая задача выполняет команду ***IRET***для возврата в предыдущую задачу, которая приводит к переключению задачи, если в регистре флагов установлен бит *NT = 1* (бит вложенности).

4. Возникло аппаратное или программное прерывание, и соответствующий элемент дескрипторной таблицы прерываний содержит шлюз задачи.

До перехода в многозадачный режим необходимо определить дескрипторы TSS, разместить сегменты TSS в адресном пространстве и правильно их инициализировать. Для работы с сегментами TSS используется альтернативное именование. При этом создается сам дескриптор TSS и дескриптор сегмента данных, описывающий ту же область памяти. Эти два дескриптора рекомендуется располагать в GDT один за одним. Перед первым запуском задачи в сегменте TSS (в регистре CS : EIP) должен содержаться адрес первой команды, а сегментные регистры данных должны содержать селекторы сегментов данных для данной задачи. В регистр SS следует загрузить селектор сегмента стека с правильным уровнем привилегий. Регистры общего назначения могут содержать нули, если не определять конкретные начальные значения. Если задача имеет LDT, то должен быть определен селектор для этой таблицы, а если будет поддерживаться страничное преобразование, то и регистр CR3. Страничное преобразование и условие работы с математическим сопроцессором являются обязательными для всех задач.

В регистре задачи TR находится селектор дескриптора сегмента состояния задачи. Поэтому вначале переключения задачи процессор знает, куда ему нужно сохранить текущее состояние задачи, а затем в регистр TR новой задачи устанавливается бит занятости новой задачи и устанавливается бит TS задачи в регистре CR0. Селектор новой задачи берется или из команд **JMP** и **CALL,** либо берется из шлюза задачи, если адресуется шлюз задачи. Потом загружается состояние входящей задачи из сегмента TSS и продолжается выполнение следующей команды. Бит TS используется системными правилами, с целью организации использования математического сопроцессора. Если бит TS установлен и идет обращение к сопроцессору, то ОС знает, что необходимо сохранить содержимое и установить новое значение. Уровни привилегий между собой не связаны.

**Формат шлюза задач**

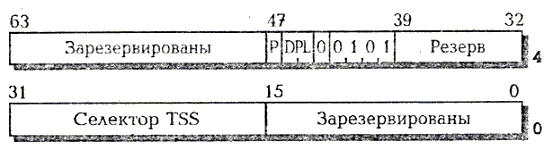
****

Рис. 9.16. Формат дескриптора шлюза задачи

Правила защиты при переключении задач выглядят так:

Max (*CPL, RPL*) ≤ *DPL*шлюза задачи

Max (*CPL, RPL*) ≤ *DPL*сегмента TSS

**.6. Общая схема переключения задач**

Дескриптор TSS

Дескриптор TSS

EAX  
EBX  
ECX  
EDX  
ESP  
EBP  
ESI

EDI

EIP

EFLAGS

CR3

CS

DS

SS

ES

FS

GS

LDT

0

31

**TR**

JMP Селектор

Смещение

0

15

**GDT**

**TSS выходящей задачи**

**TSS входящей задачи**

Рис. 9.17 Общая схема переключения задач

**Особые случаи при переключении задач**

При переключении задач могут возникнуть следующие особые случаи:

- неприсутствие;

- нарушение общей защиты;

- неверный сегмент TSS;

- нарушение стека.

Задачи являются нереентерабельными (т.е. без повторного вхождения, нельзя переключится задаче самой на себя). О занятости задачи информирует бит занятости в дескрипторе сегмента состояния задачи. Переключение на занятую задачу процессор не производит, а генерирует особый случай.

**Вложенность задач**

Когда переключение задач инициируется командой ***far call***, аппаратным прерыванием или особым случаем, задача, на которую инициируется переключение, считается вложенной в ту задачу, из которой произошло переключение. Это похоже на вызов подпрограмм.

Когда вложенная задача выполнила команду***iret***, процессор автоматически переключается на прерванную задачу. Глубина вложенности не ограничивается.

В качестве механизма связи вложенных задач используется поле обратной связи в сегменте *TSS*, в котором сохраняется старое содержимое регистра задачи. Кроме того, процессор устанавливает в единицу бит вложенной задачи *NT* в регистре *EFLAGS* , который свидетельствует о том, поле обратной связи содержит информативное значение.

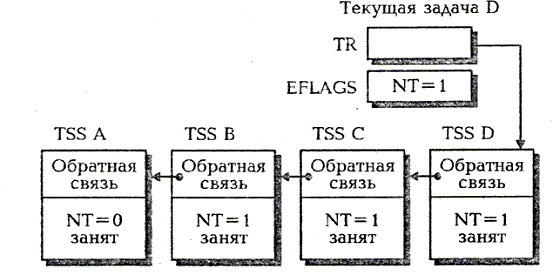


Рис. 9.18. Цепь из трех вложенных задач

Переключение на задачу А было проведено с помощью команды ***farjmp***, т.к. бит *NT=0*, а далее эта задача вызвала следующую с помощью команды ***farcall***. Биты вложенности устанавливается в 1, а поле обратной связи указывает на предыдущую задачу. Но для всех этих задач остается включенным бит занятости в дескрипторе сегмента *TSS.*

Фактически вложенность задач учитывает только команда *IRET*, обычная команда возврата из подпрограммы *RET*вложенности не учитывает.

**Двоичная карта разрешения ввода-вывода**

Последнее 16-битное слово в базовом *TSS* содержит смещение начала двоичной карты разрешения ввода-вывода. Она является дополнительным средством механизма защиты по привилегиям. Такая карта создается для каждой задачи и участвует в контроле привилегии команд ввода-вывода.

В двоичной карте каждый бит соответствует одному адресу ввода-вывода, то есть байтному порту ввода-вывода. Младший бит первого байта карты относится к нулевому адресу и далее- по возрастающей. Полная карта ввода-вывода может занимать 8 Кб памяти. Но она может занимать меньше, если не все порты нужно закрывать на разрешение доступа.

Двоичная карта ввода-вывода должна располагаться вблизи сегмента *TSS*.

**Прерывания и особые случаи**

В ходе работы ЭВМ могут возникать ситуации настолько важные, что заставят процессор приостановить текущую программу и переключиться на выполнение другой, более срочной и важной. Причинами прерывания текущей программы могут быть:

1. Внешний сигнал по входу маскируемых прерываний *INTR* или немаскируемого прерывания *NMI;*

2. Ненормальная ситуация, сложившаяся при выполнении конкретной команды и препятствующая нормальному ходу выполнения программы;

3. Находящаяся в программе команда прерывания *INTn*, где n – номер прерывания.

Реагируя на внешнее прерывание, процессор должен:

1) определить его источник;

2) сохранить минимальный контекст текущей программы (по крайней мере адрес возврата);

3) переключиться на специальную программу – обработчик прерывания, которым может быть процедура или задача. Обработчик должен выполнять действия, нормализующие ситуацию, после чего процессор возвратится к прерванной программе и она возобновиться так, будто прерывания не было.

Программные прерывания обычно называются ***особыми случаями (exeptions)***. Реакция процессора на особые случаи подобна реакции на аппаратное прерывание, а действия обработчика зависят от условий, при которых возник особый случай.

Особые случаи бывают следующих типов:

– нарушение (*fault);*

– ловушка (*trap*);

– авария (*abort*).

*Нарушение* – это такой особый случай, который процессор может обнаружить до возникновения фактической ошибки. Например, нарушение правил привилегий или выход за границы сегмента. После корректной обработки нарушения можно продолжить программу, осуществив повторное выполнение команды, вызвавшей особый случай.

*Ловушка* – это такой особый случай, который обнаруживается после окончания виновной команды. После его обработки процессор возобновляет действия с той командой, которая находится после команды, приведшей к особому случаю. Примером таких команд являются команды, вызвавшие переполнение или команда INT.

*Авария* – очень серьезная ошибка, в результате которой часть программы или вся программа теряется или искажается и продолжить ее невозможно. Причину аварии установить невозможно, поэтому restart программы не удается и программу необходимо прекратить. К авариям относятся аппаратные ошибки или серьезные искажения системных данных или системных программ.

**Прерывания и особые случаи в процессоре 8086**

Всем источникам прерываний назначается номер от 0 до 255 (**вектор**), который позволяет процессору выбрать нужный обработчик. Некоторые вектора зарезервированы для специальных целей, а незанятые вектора предоставляются в распоряжение программы.

Таблица векторов прерываний находится в памяти и состоит из 4-байтных элементов и начинается с нулевого физического адреса памяти, занимая максимально 1Кб. Любой элемент таблицы представляет собой полный указатель – селектор смещения точки входа в обработчик прерывания

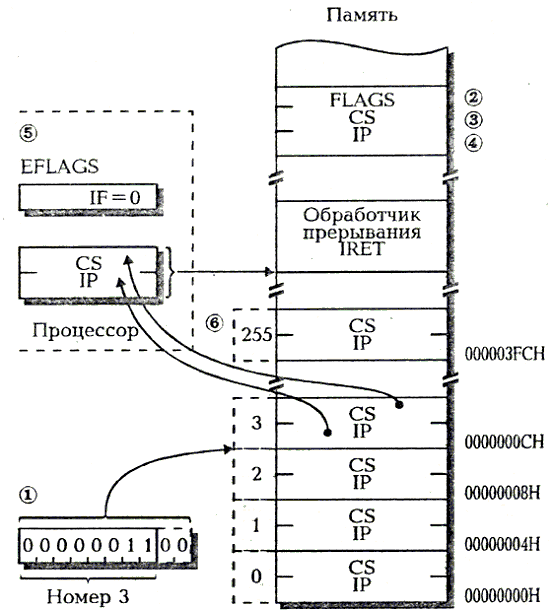


Рис. 9.19. Схема обработки прерываний при работе

процессора в реальном режиме

При обработке аппаратных прерываний выполняются следующие действия.

1. Вводится номер прерывания от программируемого контроллера прерываний. Для этого требуется два цикла шины на подтверждение прерывания. Контроллер прерываний – это микросхема, выполняющая вставку вектора.
2. Включается в стек содержимое регистра флагов (16 бит).
3. Включается в стек содержимое регистра *CS*.
4. Включается в стек содержимое регистра *IP*.
5. Сбрасывается в ноль флажок прерывания *IF*, запрещая восприятие дальнейших прерываний до момента установки в 1 этого флажка программой обработки прерывания.
6. По номеру прерывания производится обращение к соответствующему элементу таблицы векторов, из которой извлекается содержимое регистров *CS* и*IP*.
7. Начинается выполнение обработчика прерывания с точки входа, определяемое регистрами *CS:IP*.

Когда обработчик прерывания закончил свои действия, он выполняет команду возврата из прерывания (*IRET*) и управление передается прерванной подпрограмме.

При обработке команды *INTn* номер прерывания содержится в команде, соответственно не нужны циклы шины, а флажок I*F* не сбрасывается в 0, что разрешает обработку аппаратных прерываний.

***Особый случай*** - это внутреннее событие при работе процессора. Для особых случаев назначаются фиксированные номера в диапазоне от 0 до 31.

Процессор 8086 распознает только 5 особых случаев и всегда реагирует на них независимо от состояния флажков прерывания:

(0) – особый случай деления на ноль;

(1) – особый случай покомандной или пошаговой обработки;

(2) – особый случай немаскируемых прерываний NMI;

(3) – особый случай контрольной точки ***int 3;***

(4) – особый случай прерывания при переполнении (фиксируется, если при выполнении команды *into* устанавливается флаг *of=1*).

**Прерывания в защищенном режиме**

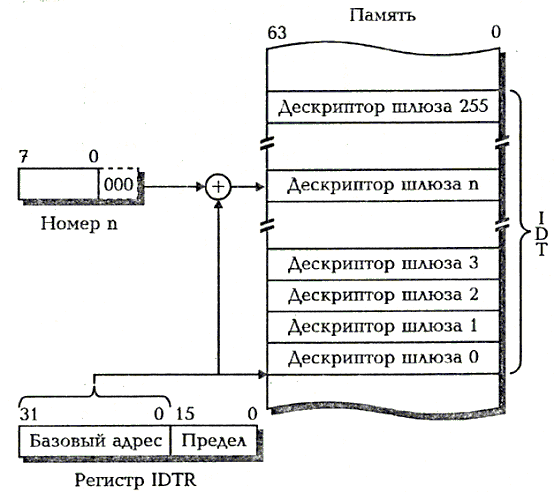


Рис. 9.20. Схема обработки прерываний при работе

процессора в защищенном режиме

Механизм обработки прерываний и особых случаев при работе в защищенном режиме усовершенствован по сравнению с реальным режимом, а именно:

1. Таблица векторов прерываний трансформирована в дескрипторную таблицу прерываний (*IDT*).

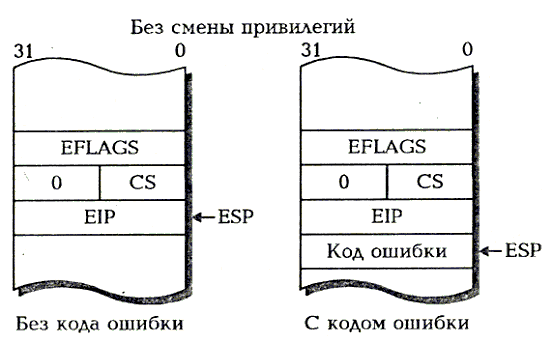
2. Усложнен процесс перехода к обработке прерывания или особых случаев.

3. Имеется возможность передачи дополнительной информации о причине возникновения особых случаев обработки прерываний.

4. В защищенном режиме процессор распознает большее число особых случаев (например: 0 – деление на нуль, 1 – пошаговая обработка, 2 – критическая точка).

Каждому аппаратному прерыванию и программному особому случаю назначается индивидуальный номер, по которому процессор обращается к дескрипторной таблице прерываний. Таблица прерываний находится в памяти и её линейный базовый адрес хранится в регистре IDTR. Она представляет собой массив шлюзов, через который осуществляется передача управления обработчику прерываний или особых случаев. В этом регистре также хранится предел, определяющий размер таблицы. Таблица прерываний является общей для всех задач и процессов. Поэтому никакой информации о ней не сохраняется в сегменте состояния задач. Программы не имеют возможность напрямую обращаться к *IDT* , т.е. они могут выбирать через селектор только *LDT*или*GDT.*

Процессор до передачи управления включает в стек обработки минимум 12 байт (адрес возврата и содержимое регистра флажков).



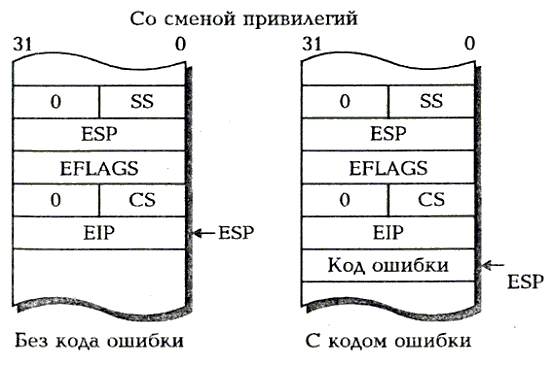
**

Рис. 9.21. Форматы стека при передаче управления

обработчику прерываний

В некоторых программных особых случаях процессор включает в стек код ошибки, по которому можно уточнить причину особого случая. Уровень привилегий обработчика определяется полем *DPL* в дескрипторе его сегмента кода. Обработчик может обращаться к любому сегменту памяти через глобальные или локальные (для текущей задачи) таблицы на своем уровне привилегий и передавать управление с памяти команд *farjmp* и *farcall*, изменять уровень привилегий с помощью шлюза вызова и производить ввод-вывод. Управление возвращается прерванной программе командой *IRET*.

Особых случаев при работе в защищенном режиме больше, чем при работе в реальном режиме. Если на границе команды возникает несколько особых случаев, то процессор обслужит их в порядке их приоритетности. Менее приоритетные особые случаи уничтожаются; менее приоритетные прерывания остаются ждать. Уничтоженные особые случаи формируются вновь, когда обработчик прерываний возвращает управление в точку прерывания.

Особые случаи имеют следующую приоритетность(в порядке убывания):

- особые случаи отладки;

- немаскируемые прерывания;

- маскируемые прерывания;

- нарушение при выборке команды и её декодировании;

- особый случай недоступности сопроцессора;

- нарушения неприсутствия сегмента;

- нарушения стека;

- нарушения общей защиты для операндов в памяти;

- нарушение выравнивания операндов в памяти;

- страничное нарушение для операндов в памяти.

**Дескрипторная таблица прерываний**

Дескрипторная таблица прерываний представляет массив восьмибайтных дескрипторов. В ней должны быть определены 256 обработчиков. Поэтому ее максимальный предел должен иметь значение 2047 или 7FFh. Если возникает особый случай и его номер выбирается за пределами таблицы, генерируется нарушение общей защиты, т. е. процессор переходит в режим отключения. В этом режиме он прекращает выполнение команд до получения немаскируемых прерываний или при сбросе процессора. Переход в этот режим процессор показывает специальным циклом шины и к этому циклу можно прикрепить какой-либо аппаратный индикатор.

В отличие от глобальной таблицы, нулевой дескриптор является действительным.

В дескрипторной таблице прерываний могут храниться три вида дескрипторов:

- шлюз задачи;

- шлюз прерывания;

- шлюз ловушки.

Зарезервированы PDPL 0 0 1 0 1 Резерв

Селектор TSS Зарезервированы

31 15 0

4

0





*Шлюз задачи*

Смещение 31-16 PDPL 0 0 1 0 1 0 0 0 Резерв

Селектор Смещение 15-0

31 15 0

4

0

15 47 39 32

*Шлюз прерывания*

15 47 39 32

Смещение 31-16 PDPL 0 1 1 1 1 0 0 0 Резерв

4

31 15 0

0

Селектор Смещение 15-0

*Шлюз ловушки*

Рис. 9.22. Форматы дескрипторов шлюзов, разрешенных в

дескрипторной таблице прерываний

Шлюзы ловушки и прерывания очень похожи на шлюз вызова, за исключением того, что в них нет поля счетчика слов, т. к. обработчикам прерываний параметры не передаются. Эти шлюзы однозначно определяют точку входа в обработчик. Поле уровня привилегий *DPL* в этих дескрипторах определяет тот минимальный уровень, который необходим для передачи управления через шлюз. Рекомендуется это поле устанавливать равным 3, чтобы дескрипторы из таблицы прерываний были доступны всем задачам, выполняемым в системе.

**Шлюз ловушки**

Когда возникает особый случай и по его номеру выбирается шлюз ловушки, процессор сохраняет часть своего состояния в соответствие с приведенным ранее рисунком, а затем осуществляет межсегментную передачу управления, используя селектор и смещение из шлюза ловушки.

Селектор должен выбирать сегмент кода, который может адресовать через *GDT* или *LDT*. Но рекомендуется хранить дескрипторы всех обработчиков в *GDT*.

Целевой сегмент кода может быть отмечен как отсутствующий. В этом случае вырабатывается особый случай неприсутствия, по которому должно быть выполнено перемещение обработчика из внешней памяти в оперативную. По коду*IRET* процессор извлекает из стека адрес возврата и содержимое регистра флажков, а, если происходит смена привилегий, то и адрес внешнего стека, после чего прерванная задача возобновляется.

**Шлюз прерывания**

Действует, как и шлюз ловушки, но в этом случае процессор сбрасывает в ноль флажки прерываний. Этот сброс осуществляется после включения регистра флажков в стек, но до выполнения первой команды обработчика. Это значит, что блокируются вновь поступившие аппаратные прерывания до тех пор, пока текущий обработчик не завершит свою работу. Но немаскируемые и программные прерывания не запрещаются.

**Шлюз задачи**

Когда номер прерывания выбирает в таблице прерываний шлюз задачи, процессор переключается на новую задачу, определенную селектором сегмента *TSS* задачи в шлюзе задачи. Такой шлюз не может содержать селектор сегмента*TSS* из *LDT*и селектор шлюза еще одной задачи.

При переключении задачи селектор *TSS* текущие задачи помещает в поле обратной связи *TSS* новой задачи и в ее регистре флажков устанавливает *NT = 1.* Когда обработка завершается командой *iret*осуществляется переключение на прерванную задачу с использованием поля обратной связи.

Обработка особых случаев через шлюз задачи имеет следующие преимущества:

- автоматически сохраняется весь контекст прерванной задачи;

- обработчик особых случаев не может исказить прерванную задачу, так как она полностью изолирован от нее;

- обработчик прерываний может работать на любом уровне привилегий в заведомо правильной среде. Может иметь свое локальное адресное пространство, благодаря LDT.

Недостатки применения шлюза задачи:

- реакция процессора несколько замедлена, т.к. необходимо сохранять большее количество информации при переключении задач;

- в шлюзе задачи невозможно определить начальную точку выполнения задачи;

- сложно получить информацию о прерванной задаче. Эту информацию можно получить, используя только поле обратной связи.

Когда для обработки особых случаев привлекаются шлюзы задач, необходимо обратить внимание на то, чтобы избежать рекурсии особых случаев, т.к. и задача-обработчик, и задача, которая была прервана, будут отмечены как занятые, а на занятые задачи переключиться нельзя.

# Практический раздел

## Индивидуальная практическая работа №1

### Система команд, файловая структура, работа с файлами и управление ОС Unix/Linux с помощью интерпретатора BASH)

***Цель работы: – изучение команд ОС для работы с файлами, каталогами, дисками, системной датой и временем; текстового редактора Kate и файлового менеджера MidnightCommander, исследовать основные объекты, команды, типы данных и операторы управления интерпретатора BASH; создать скрипт-файл.***

### Теоретическая часть

Операционная система ОС ***Linux***создана на основе ОС ***UNIX*** и во многом имеет схожую структуру и систему команд. Пользователь может работать в текстовом режиме с помощью командной строки, или с использованием графического интерфейса ***XWindow***и одного из менеджеров рабочего стола (например, ***KDE***или ***GNOME***). Причем, одновременно в системе могут работать 7 пользователей (6- в текстовом режиме консоли и 1 – в графическом режиме).

В табл. 1 приведены основные команды системы

Таблица 1

|  |  |  |  |
| --- | --- | --- | --- |
| Команда | Аргументы/ключи | Пример | Описание |
| ***dir*** | каталог | ***dir***  ***dir /home*** | Выводит на консоль содержимое каталога |
| ***ls*** | ***-all***  и другие (см. man) | ***ls -all*** | Выводит на консоль содержимое каталога |
| ***ps*** | ***-a***  ***-x*** и другие (см. man) | ***ps -a*** | Выводит на консоль список процессов |
| ***mkdir*** | имя каталога | ***mkdir stud11*** | Создает каталог |
| ***rmdir*** | имя каталога | ***rmdir stud11*** | Удаляет каталог |
| ***rm*** | файл | ***rm myfile1*** | Удаляет файл |
| ***mv*** | файлновое\_имя | ***mv myfile1 myf1*** | Переименование файла |
| ***cat*** | файл | ***cat 1.txt*** | Вывод файла на консоль |
| ***cd*** | имя каталога | ***cd home*** | Переход по каталогам |
| ***grep*** | (см. man) | ***grep "^a" "words.txt"*** | Поиск строки в файле |
| ***kill*** | ***pid***процесса | ***kill 12045*** | Уничтожает процесс  посылкой сигнала |
| ***top*** |  |  | Выводит на консоль список процессов |
| ***htop*** |  |  | Выводит на консоль полный список запущенных процессов |
| ***su*** |  |  | Переход в режим root |
| ***chmod*** | права\_доступа файл | ***chmod 777 1.txt*** | Изменение прав доступа к файлам |
| ***mount*** | устройство каталог | ***mount /dev/cdrom /MyCD*** | Монтирование устройств |
| ***dd*** | ***if=***файл***of=***файл***bs=n count=n*** | ***dd if=/dev/hda1***  ***of=/F.bin bs=512 count=1*** | Копирование побайтное |
| ***ln*** | файл1 файл2  ***-l*** | ***ln файл1 файл2***  ***ln –l файл1 файл2*** | Создать жёсткую или символическую ссылку на файл |
| ***uname*** | ***-a*** | ***uname -a*** | Информация о системе |
| ***find*** | ***find***файл | ***find /home a1.txt*** | Поиск файлов |
| ***man*** |  | ***man fgetc*** | Справка по системе |
| ***info*** |  | ***info fgetc*** | Справка по системе |

***Linux*** и ***Windows*** используют различные файловые системы для хранения и организации доступа к информации на дисках. В ***Linux*** используются файловые системы-***Ext2/Ext3***, ***RaiserFS*** и другие. Все современные файловые системы имеют поддержку ***журналирования***. ***Журналируемая*** файловая система сначала записывает изменения, которые она будет проводить в отдельную часть файловой системы (***журнал***) и только потом вносит необходимые изменения в остальную часть файловой системы. После удачного выполнения всех транзакций, записи удаляются из ***журнала***. Это обеспечивает лучшее сохранение целостности системы и уменьшает вероятность потери данных. Следует отметить, что ***Linux*** поддерживает доступ к ***Windows***-разделам.

Файловая система ***Linux*** имеет лишь один корневой каталог, который обозначается косой чертой (***/***). В файловой структуре ***Linux*** нет дисков ***A, B, C, D,*** а есть только каталоги. Различаются прописные и строчные буквы в командах, именах файлов и каталогов. В ***Windows*** у каждого файла существует лишь одно имя, в ***Linux*** их может быть много. Это – «***жесткие***» ссылки, которые указывают непосредственно на индексный дескриптор файла. Жесткая ссылка – это один из принципов организации файловой системы ***Linux***.

Для выполнения операций записи и чтения данных в существующем файле его следует открыть:

***int open (const char \*pathname, int flags, [mode\_t mode]);***

***int fopen (const char \*pathname, int flags, [mode\_t mode]);***

Второй аргумент ***flags*** функции ***open ()*** имеет целочисленный тип и определяет метод доступа и принимает одно из значений, заданных константами в заголовочном файле ***fcnt1.h***. В файле определены три постоянных:

***O\_RDONLY*** – открыть файл только для чтения,

***O\_WRONLY*** – открыть файл только для записи,

***O\_RDWR*** – открыть файл для чтения и записи,

Для***fopen()***- ***“r”***, ***“w”***, ***“rw”.***

Третий параметр ***mode***устанавливает права доступа к файлуи является необязательным, он используется только вместе с флагом ***O\_CREAT***.Примерсозданияновогофайла:

***# include <sys / types.h>***

***# include <sys / stat.h>***

***# include <fcnt1.h>***

***int Fd1;***

***FILE \*F1;***

***F1=fopen (“Myfile2.txt”, “w”, 644);***

***Fd1=open (“Myfile1.txt”, O\_CREAT, 644);***

Системные вызовы ***sta****t* и ***fstat*** позволяют процессу определить значения свойств в существующем файле.

***#include <sys/types.h>***

***#include <sys/stat.h>***

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

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

Где***pathname*** – полное имя файла, ***buf*** –структура типа ***stat***. Эта структура после успешного вызова будет содержать связанную с файлом информацию.

Поля структуры stat включает следующие элементы:

***structstat {***

***dev\_tst\_dev; /\**** логическое устройство, где находится файл ***\*/***

***ino\_tst\_ino; /\****номер индексного дескриптора ***\*/***

***mode\_tst\_mode; /\**** права доступа к файлу***\*/***

***nlink\_t st\_nlink; /\**** количество жестких ссылок на файл ***\*/***

***uid\_t st\_uid; /\**** ID пользователя-владельца ***\*/***

***gid\_t st\_gid; /\**** ID группы-владельца ***\*/***

***dev\_t st\_rdev; /\**** тип устройства ***\*/***

***off\_t st\_size; /\**** общий размер в байтах ***\*/***

***unsigned long st\_blksize; /\**** размер блока ввода-вывода ***\*/***

***unsigned long st\_blocks; /\**** число блоков, занимаемых файлом ***\*/***

***time\_t st\_atime; /\**** время последнего доступа ***\*/***

***time\_t st\_mtime; /\**** время последней модификации ***\*/***

***time\_t st\_ctime; /\**** время последнего изменения ***\*/***

***};***

Права доступа в ***Linux***. Права доступа к файлам представлены в виде последовательности бит, где каждый бит означает разрешение на запись (***w***), чтение (***r***) или выполнение (***x***). Права доступа записываются для владельца-создателя файла(***owner***); группы, к которой принадлежит владелец–создатель файла(***group***); и всех остальных (***other***). Например, при выводе команды ***dir***запись типа:

-***rwxr-xr-w1.exe***

означает, что владелец файла ***1.exe***имеет права на чтение, запись и выполнение, группа имеет права только на чтение и выполнение, все остальные имеют права только на чтение. В восьмеричном виде получится значение ***0754***. В действительности манипулирует файлами не сам пользователь, а запущенный им процесс. Для просмотра прав доступа можно использовать функцию ***stat***.

Пример: stat***(“1.exe”, &st1);***

Для записи прав доступа служит функция ***chmod***:

***#include <sys/types.h>***

***#include <sys/stat.h>***

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

Пример: chmod***(“1.exe”, 0777);***

Структура каталогов ОС ***Linux*** представлена в табл. 2. Есть также несколько полезных сокращений для имен каталогов:

* Одиночная точка (**.**) обозначает текущий рабочий каталог.
* Две точки (**..**) обозначают родительский каталог текущего рабочего.
* Тильда (**~**) обозначает домашний каталог пользователя (обычно это каталог, который является текущим рабочим при запуске Bash).

Таблица 2

|  |  |
| --- | --- |
| ***/*** | Корневой каталог |
| ***/bin*** | Содержит исполняемые файлы самых необходимых для работы системы программ. Каталог ***/bin*** не содержит подкаталогов. |
| ***/boot*** | Здесь находятся само ядро системы (файл ***vmlinuz***-...) и файлы, необходимые для его загрузки. |
| ***/dev*** | Каталог ***/dev*** содержит файлы устройств (драйверы). |
| ***/etc*** | Это каталог конфигурационных файлов, т. е. файлов, содержащих информацию о настройках системы (например, настройки программ). |
| ***/home*** | Содержит домашние каталоги пользователей системы. |
| ***/lib*** | Здесь находятся библиотеки (функции, необходимые многим программам). |
| ***/media*** | Содержит подкаталоги, которые используются как точки монтирования для сменных устройств (CD-ROM'ов, floppy-дисков и др.) |
| ***/mnt*** | Данный каталог (или его подкаталоги) может служить точкой монтирования для временно подключаемых файловых систем. |
| ***/proc*** | Содержит файлы с информацией о выполняющихся в системе процессах. |
| ***/root*** | Это домашний каталог администратора системы. |
| ***/sbin*** | Содержит исполняемые программы, как и каталог ***/bin***. Однако использовать программы, находящиеся в этом каталоге может только администратор системы (***root***). |
| ***/tmp*** | Каталог для временных файлов, хранящих промежуточные данные, необходимых для работы тех или иных программ, и удаляющиеся после завершения работы программ. |
| ***/usr*** | Каталог для большинства программ, которые не имеют значения для загрузки системы. Структура этого каталога фактически дублирует структуру корневого каталога. |
| ***/var*** | Содержит данные, которые были получены в процессе работы одних программ и должны быть переданы другим, и файлы журналов со сведениями о работе системы. |

***Bash*** - это ***sh***-совместимый интерпретатор командного языка, выполняющий команды, прочитанные со стандартного входного потока или из файла. ***Скрипт-файл*** – это обычный текстовый файл, содержащий последовательность команд ***bash***, для которого установлены права на выполнение. Пример скрипта, выводящего содержимое текущего каталога на консоль и в файл:

***#!/bin/bash***

***dir***

***dir> 1.txt***

Следующие переменные используются командным интерпретатором:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ***$0,$1,S2,$3…*** | | | Значения аргументов командной строки при запуске скрипта.Где ***$0***-имя самого файла скрипта, ***$1***- первый аргумент, ***$2***- второй аргумент, и т.д. | |
| ***$@*** | | | Все аргументы командной строки, каждый в кавычках | |
| ***$?*** | | Код возврата последней команды | | | |
|  |  | | |
|  |  | | |

Пример простого файла-скрипта***mydir.sh***, выводящего на консоль и в файл содержимое каталога,где имя каталога передаётся скрипту в качестве первого аргумента командной строки:

Содержимое файла ***mydir.sh***:

***#!/bin/bash***

***dir $1***

***dir $1 > 1.txt***

Длязапускаскриптанеобходимо изменить его права доступа, разрешив выполнение:

>***chmod 777mydir.sh***

Запускскрипта:

***>./mydir.sh/home/stud***

Можно создать собственную переменную и присвоить ей значение:

***A=121***

***A=”121”***

***letA=121***

***let “A=А+1”***

Вывод значения на консоль:

***echo $A***

Проверкаусловия: ***test[expr]***

где ***expr***: а) для строк: ***S1 = S2*** ***S1***содержит***S2***

***S1 != S2******S1***несодержит***S2***

***-nS1*** если длина ***S1 >0***

***-zS1*** если длина ***S1 =0***

б) целые***i1***и***i2***

***i1 – ge i2***

***i1 – gt i2***

***i1 – ie i2***

***i1 – et i2***

***i1 – nti2***

в) файлы

***-dname\_file*** является ли файл каталогом

***-fname\_file*** является ли файл обычным файлом

***-rname\_file*** доступен ли файл для чтения

***-sname\_file*** имеет ли файл ненулевую длину

***-wname\_file*** доступен ли файл для записи

***-xname\_file*** является ли файл исполняемым

г) логически операции

***!exp*** логическое отрицание (не)

***exp1 –aexp2*** умножение условий (и)

***exp1 –oexp2*** сложение условий (или)

Проверка условия: ***if [expr ]***

***then com 1*** Если условие ***expr=true***то команда

***… com 1…com n***

***com n***

***(elif expr2***

***com1***

***…***

***comn***

***)***

***else***

***com 1***

***…***

***comn***

***fi***

Проверка нескольких условий:

***casestring1 in***

***str 1)***

***com 1***

***…***

***com n***

***; ;***

***str 2)***

***com 1***

***…***

***com n***

***; ;***

***str 3)***

***com 1***

***…***

***com n***

***; ;***

***\*)***  // default

***com 1***

***…***

***com n***

***; ;***

***esac***

Функция пользователя: ***fname2 (arg1, arg2...argN)***

***{  
commands  
}***

Организация циклов:

1. ***for var1 in list***

***do***

***com1***

***…***

***com n***

***done***

2.***whileexp***

***com1***

***…***

***comn***

***end***

3.***untilexp // аналогdo-while***

***do***

***com1***

***…***

***com n***

***done***

### Варианты индивидуальных заданий

**Во всех заданиях** д**олжен быть контроль ошибок (**если к какому-либо каталогу нет доступа, необходимо вывести соответствующее сообщение и продолжить выполнение).

Вывод сообщений об ошибках должен производиться в стандартный поток вывода сообщений об ошибках (***stderr***) в следующем виде:

***имя\_модуля: текст\_сообщения***.

Пример:

***./1.exe :Erroropenfile: 1.txt***

Имя модуля, имя файла берутся из аргументов командной строки.

Номер варианта индивидуального задания K = (Номер Вашего паспорта) mod 7.

Где номер паспорта: Открываете паспорт и смотрите номер (только номер, без серии)

Например: №=1234567

(1234567) mod 7 =5

(1234567) mod 7 --- значит 1234567 по модулю 7 (или остаток отделения на 7)

1. Написать скрипт для поиска файлов заданного размера в заданном каталоге (имя каталога задаётся пользователем в качестве третьего аргумента командной строки). Диапазон (мин.- мах.) размеров файлов задаётся пользователем в качестве первого и второго аргумента командной строки. Проверить работу программы для каталога***/usr и*** диапазона (мин.- мах.)***1000 1010.***
2. Написать скрипт с использованием цикла ***for***, выводящий на консоль размеры и права доступа для всех файлов в заданном каталоге и всех его подкаталогах (имя каталога задается пользователем в качестве первого аргумента командной строки). Проверить работу программы для каталога***/usr.***
3. Написать скрипт для поиска заданной пользователем строки во всех файлах заданного каталога и всех его подкаталогов (строка и имя каталога задаются пользователем в качестве первого и второго аргумента командной строки). На консоль выводятся полный путь и имена файлов, в содержимом которых присутствует заданная строка, и их размер. Если к какому-либо каталогу нет доступа, необходимо вывести соответствующее сообщение и продолжить выполнение. Проверить работу программы для каталога***/usr***найти строку «***stdio.h»***.
4. Написать скрипт поиска одинаковых по их содержимому файлов в двух каталогов, например, ***Dir1*** и ***Dir2***. Пользователь задаёт имена ***Dir1*** и ***Dir2***в качестве первого и второго аргумента командной строки. В результате работы программы файлы, имеющиеся в ***Dir1***, сравниваются с файлами в ***Dir2*** по их содержимому. На экран выводятся число просмотренных файлов и результаты сравнения. Проверить работу программы для каталога***/usr (Dir1)*** и любого Вашего каталога в каталоге***/home (Dir2).***
5. Написать скрипт находящий в заданном каталоге и всех его подкаталогах все файлы, владельцем которых является заданный пользователь. Имя владельца и каталог задаются пользователем в качестве первого и второго аргумента командной строки. Скрипт выводит результаты в файл (третий аргумент командной строки) в виде полный путь, имя файла, его размер. На консоль выводится общее число просмотренных файлов. Проверить работу программы для каталога***/usr***пользователь***root***.
6. Написать скрипт находящий в заданном каталоге и всех его подкаталогах все файлы, заданного размера в заданном каталоге (имя каталога задаётся пользователем в качестве первого аргумента командной строки). Диапазон (min.- max.) размеров файлов задаётся пользователем в качестве второго и третьего аргументов командной строки. Cкрипт выводит результаты поиска в файл (четвертый аргумент командной строки) в виде полный путь, имя файла, его размер. На консоль выводится общее число просмотренных файлов. Проверить работу программы для каталога***/usr и*** диапазона (мин.- мах.)***1000 1010.***
7. Написать скрипт подсчитывающий суммарный размер файлов в заданном каталоге и всех его подкаталогах (имя каталога задаётся пользователем в качестве аргумента командной строки). Скрипт выводит результаты подсчёта в файл (второй аргумент командной строки) в виде каталог (полный путь), суммарный размер файлов число просмотренных файлов. Проверить работу программы для каталога***/usr.***

## Индивидуальная практическая работа №2

### Работа с каталогами, основные принципы программирования процессов и потоков в ОС UNIX/LINUX.

Цель работы –изучение файловой системы ОС Linux и основных функций для работы с каталогами и файлами, исследовать методы создания процессов в ОС Linux, основные функции создания и управления процессами, обмен данными между процессами.

### Теоретическая часть

Каталоги в ОС ***Linux***– это особые файлы. Для открытия или закрытия каталогов существуют вызовы:

***#include <dirent.h>***

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

***intclosedir( DIR \*dirptr);***

Для чтения записей каталога существует вызов:

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

Структура dirent такова:

***struct dirent {***

***longd\_ino; //*** индексныйдескрипторфайла

***off\_td\_off;//***смещение данного элемента в реальном каталог

***unsignedshortd\_reclen;//*** длинаструктуры

***chard\_name [1]; //*** имяэлементакаталога

***};***

Поле ***d\_name*** есть начало массива символов, задающего имя элемента каталога. Данное имя ограничено нулевым байтом и может содержать не более ***MAXNAMLEN*** символов.

Примервызова:

***DIR \*dp;***

***struct dirent \*d;***

***d=readdir(dp);***

При первом вызове функция ***readdir*** в структуру***d*** будет считана первая запись каталога. После прочтения последней записи каталога будет возвращено значение ***NULL***. Для возврата указателя в начало каталога на первую запись существует вызов:

***voidrewindir(DIR \*dirptr);***

Чтобы получить текущий рабочий каталог (путь) существует функция:

***char \*getcwd(char \*name, size\_t size);***

В ОС Linuxдля создания процессов используется системный вызов ***fork ()****:*

***pid\_tfork (void);***

В результате успешного вызова ***fork ()*** ядро создаёт новый процесс, который является почти точной копией вызывающего процесса. Другими словами, новый процесс выполняет копию той же программы, что и создавший его процесс, при этом все его объекты данных имеют те же самые значения, что и в вызывающем процессе.

Созданный процесс называется ***дочерним процессом***, а процесс, осуществивший вызов ***fork ()***, называется ***родительским***. После вызова родительский процесс и его вновь созданный потомок выполняются одновременно, при этом оба процесса продолжают выполнение с оператора, который следует сразу же за вызовом ***fork ()***. Процессы выполняются в разных адресных пространствах, поэтому прямой доступ к переменным одного процесса из другого процесса невозможен.

Следующая короткая программа более наглядно показывает работу вызова ***fork ()***и использование процесса:

***#include <stdio.h>***

***#include <unistd.h>***

***intmain ()***

***{***

***pid\_tpid; /\* идентификатор процесса \*/***

***printf (“Пока всего один процесс\n”);***

***pid = fork (); /\*Создание нового процесса \*/***

***printf (“Ужедва процесса\n”);***

***if (pid = = 0)***

***{***

***printf (“Это Дочерний процесс его pid=%d\n”, getpid());***

***printf (“А pidего Родительского процесса=%d\n”, getppid());***

***}***

***else if (pid > 0)***

***printf (“Это Родительский процессpid=%d\n”, getpid());***

***else***

***printf (“Ошибка вызова fork, потомок не создан\n”);***

***}***

Для корректного завершения дочернего процесса в родительском процессе необходимо использовать функцию ***wait()***или***waitpid()***:

***pid\_t wait(int \*status);  
pid\_t waitpid(pid\_t pid, int \*status, int options);***

Функция ***wait()*** приостанавливает выполнение родительского процесса до тех пор, пока дочерний процесс не прекратит выполнение или до появления сигнала, который либо завершает текущий процесс, либо требует вызвать функцию-обработчик. Если дочерний процесс к моменту вызова функции уже завершился (так называемый ***«зомби»***), то функция немедленно возвращается. Системные ресурсы, связанные с дочерним процессом, освобождаются.

Функция ***waitpid()*** приостанавливает выполнение родительского процесса до тех пор, пока дочерний процесс, указанный в параметре ***pid***, не завершит выполнение, или пока не появится сигнал, который либо завершает родительский процесс, либо требует вызвать функцию-обработчик. Если указанный дочерний процесс к моменту вызова функции уже завершился (так называемый ***«зомби»***), то функция немедленно возвращается. Системные ресурсы, связанные с дочерним процессом, освобождаются. Параметр ***pid*** может принимать несколько значений:

***pid<-1***означает, что нужно ждать любого дочернего процесса, чей идентификатор группы процессов равен абсолютному значению *pid*.

***pid=-1***означает ожидать любого дочернего процесса; функция **wait** ведет себя точно так же.

***pid =0***означает ожидать любого дочернего процесса, чей идентификатор группы процессов равен таковому у текущего процесса.

***pid>0***означает ожидать дочернего процесса, чем идентификатор равен ***pid***.

Значение ***options*** создается путем битовой операции ***ИЛИ*** над следующими константами:

**WNOHANG**- означает вернуть управление немедленно, если ни один дочерний процесс не завершил выполнение.

**WUNTRACED -** означает возвращать управление также для остановленных дочерних процессов, о чьем статусе еще не было сообщено.

Каждый дочерний процесс при завершении работы посылает своему процессу-родителю специальный сигнал ***SIGCHLD***, на который у всех процессов по умолчанию установлена реакция "игнорировать сигнал". Наличие такого сигнала совместно с системным вызовом ***waitpid()*** позволяет организовать асинхронный сбор информации о статусе завершившихся порожденных процессов процессом-родителем.

Для перегрузки исполняемой программы можно использовать функции семейства ***exec***. Основное отличие между разными функциями в семействе состоит в способе передачи параметров.

***int execl(char \*pathname, char \*arg0, arg1, ..., argn, NULL);***

***int execle(char \*pathname, char \*arg0, arg1, ..., argn, NULL, char \*\*envp);***

***int execlp(char \*pathname, char \*arg0, arg1, ..., argn, NULL);***

***int execlpe(char \*pathname, char \*arg0, arg1, ..., argn, NULL, char \*\*envp);***

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

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

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

***int execvpe(char \*pathname, char \*argv[],char \*\*envp);***

Существует расширенная реализация понятия ***процесс***, когда ***процесс*** представляет собой совокупность выделенных ему ресурсов и набора ***нитей исполнения***. ***Нити(threads)***или потоки процесса разделяют его программный код, глобальные переменные и системные ресурсы, но каждая ***нить*** имеет собственный программный счетчик, свое содержимое регистров и свой стек. Все глобальные переменные доступны в любой из дочерних нитей. Каждая нить исполнения имеет в системе уникальный номер – идентификатор***нити***. Поскольку традиционный процесс в концепции нитей исполнения трактуется как процесс, содержащий единственную ***нить*** исполнения, мы можем узнать идентификатор этой ***нити*** и для любого обычного процесса. Для этого используется функция ***pthread\_self()***. Нить исполнения, создаваемую при рождении нового процесса, принято называть **начальной** или **главной**нитью исполнения этого процесса. Для создания нитей используется функция ***pthread\_create***:

***#include<pthread.h>***

***int pthread\_create(pthread\_t \*thread, const pthread\_attr\_t \*attr,***

***void \*(\*start\_routine)( void\*),void \*arg);***

Функция создает новую нить в которой выполняется функция пользователя ***start\_routine***,передавая ей в качестве аргумента параметр ***arg***. Если требуется передать более одного параметра, они собираются в структуру, и передается адрес этой структуры.При удачном вызове функция ***pthread\_create*** возвращает значение ***0*** и помещает идентификатор новой нити исполнения по адресу, на который указывает параметр ***thread***. В случае ошибки возвращается положительное значение, которое определяет код ошибки, описанный в файле ***<errno.h>***. Значение системной переменной ***errno*** при этом не устанавливается.Параметр ***attr*** служит для задания различных атрибутов создаваемой нити.Функция нити должна иметь заголовок вида:

***void \* start\_routine (void \*)***

Завершение функции потока происходит если:

* функция нити вызвала функцию ***pthread\_exit()***;
* функция нити достигла точки выхода;
* нить была досрочно завершена другойнитью.

Функция ***pthread\_join()***используется для перевода нити в состояние ожидания:

***#include <pthread.h>***

***int pthread\_join (pthread\_t thread, void \*\*status\_addr);***

Функция ***pthread\_join()***блокирует работу вызвавшей ее нити исполнения до завершения нити с идентификатором ***thread***. После разблокирования в указатель, расположенный по адресу ***status\_addr***, заносится адрес, который вернул завершившийся ***thread*** либо при выходе из ассоциированной с ним функции, либо при выполнении функции ***pthread\_exit()***. Если нас не интересует, что вернула нам нить исполнения, в качестве этого параметра можно использовать значение ***NULL***.

Для компиляции программы с нитями необходимо подключить библиотеку***pthread.lib***следующим способом:

***gcc1.c –o 1.exe -lpthread***

Время в ***Linux*** отсчитывается в секундах, прошедшее с начала этой эпохи (***00:00:00 UTC, 1 Января 1970 года***). Для работы с системным временем можно использовать следующие функции:

***#include<***[***sys/time.h***](file://usr/include/sys/time.h)***>***

***time\_ttime (time\_t \*tt); //***текущее время в секундах с 01.01.1970

***struct tm \* localtime(time\_t \*tt)***

***int gettimeofday(struct timeval \*tv, struct timezone \*tz);***

***struct timeval {***

***long tv\_sec;*** /\* секунды \*/

***long tv\_usec;*** /\* микросекунды \*/

***};***

***struct tm {***

***int tm\_sec;*** /\* seconds \*/

***int tm\_min;*** /\* minutes \*/

***int tm\_hour;*** /\* hours \*/

***int tm\_mday;*** /\* day of the month \*/

***int tm\_mon;*** /\* month \*/

***int tm\_year;*** /\* year \*/

***int tm\_wday;*** /\* day of the week \*/

***int tm\_yday;*** /\* day in the year \*/

***int tm\_isdst;*** /\* daylight saving time \*/

***};***

### Варианты индивидуальных заданий

Номер варианта индивидуального задания K = (Номер Вашего паспорта) mod 19.

***Должен быть контроль ошибок для всех операций с файлами и каталогами***.

1. Отсортировать в заданном каталоге (аргумент 1 командной строки) и во всех его подкаталогах файлы по следующим критериям (аргумент 2 командной строки, задаётся в виде целого числа):1 – по размеру файла, 2 – по имени файла. Записать отсортированные файлы в новый каталог (аргумент 3 командной строки). Процедуры копирования должны запускаться в отдельном процессе для каждого копируемого файла с использованием функций ***read ()*** и ***write ()***. Каждый процесс выводит на экран свой ***pid,***полный путь, имя копируемого файла и число скопированных байт. Число запущенных процессов в любой моментвремени не должно превышать ***N*** (вводится пользователем). Проверить работу программы для каталога***/usr/includeN=6.***
2. Написать программу синхронизации двух каталогов, например, ***Dir1*** и ***Dir2***. Пользователь задаёт имена ***Dir1*** и ***Dir2***в качестве первого и второго аргумента командной строки. В результате работы программы файлы, имеющиеся в ***Dir1***, но отсутствующие в ***Dir2***, должны скопироваться в ***Dir2*** вместе с правами доступа. Процедуры копирования должны запускаться в отдельном процессе для каждого копируемого файла с использованием функций ***read ()*** и ***write ()***. Каждый процесс выводит на экран свой ***pid,*** полный путь, имя копируемого файла и число скопированных байт. Число запущенных процессов в любой моментвремени не должно превышать ***N*** (вводится пользователем). Проверить работу программы для каталога ***/usr/include/ и любого другого каталога в /home/N=6***..
3. Найти в заданном каталоге (аргумент 1 командной строки) и всех его подкаталогах заданный файл (аргумент 2 командной строки). Вывести на консоль полный путь к файлу имя файла, его размер, дату создания, права доступа, номер индексного дескриптора. Вывести также общее количество просмотренных каталогов и файлов. Процедура поиска для каждого подкаталога должна запускаться в отдельном процессе. Каждый процесс выводит на экран свой ***pid,***по лный путь, имя и размер просмотренного файла, общее число просмотренных файлов в подкаталоге. Число запущенных процессов в любой момент времени не должно превышать ***N*** (вводится пользователем). Проверить работу программы для каталога***/usr***найти файл ***stdio.hN=6***.
4. Для заданного каталога (аргумент 1 командной строки) и всех его подкаталогов вывести в заданный файл (аргумент 2 командной строки) и на консоль имена файлов, их размер и дату создания, удовлетворяющих заданным условиям: 1 – размер файла находится в заданных пределах от ***N1*** до ***N2*** (***N1,N2***задаются в аргументах командной строки), 2 – дата создания находится в заданных пределах от ***M1*** до ***M2*** (***M1***,***M2***задаются в аргументах командной строки).Процедура поиска для каждого подкаталога должна запускаться в отдельном процессе. Каждый процесс выводит на экран свой ***pid,*** полный путь, имя и размер просмотренного файла, общее число просмотренных файлов в подкаталоге. Число запущенных процессов в любой момент времени не должно превышать ***N*** (вводится пользователем). Проверить работу программы для каталога ***/usr***/ ***размер31000 31500 дата с 01.01.1970 по текущую датуN=6***.
5. Тоже что и п.8, но включая подкаталоги. Проверить работу программы для каталога: /***usr/***размер ***31000 31500N=6***.
6. Подсчитать суммарный размер файлов в заданном каталоге (аргумент 1 командной строки) и для каждого его подкаталога отдельно. Вывести на консоль и в файл (аргумент 2 командной строки) название подкаталога, количество файлов в нём, суммарный размер файлов, имя файла с наибольшим размером. Процедура просмотра для каждого подкаталога должна запускаться в отдельном процессе. Каждый процесс выводит на экран свой ***pid,*** полный путь, имя и размер просмотренного файла, общее число просмотренных файлов в подкаталоге. Число запущенных процессов в любой момент времени не должно превышать ***N*** (вводится пользователем). Проверить работу программы для каталога ***/usr N=6.***
7. Написать программу, находящую в заданном каталоге и всех его подкаталогах все исполняемые файлы. Диапазон (мин. мах.) размеров файлов задаётся пользователем в качестве первого и второго аргумента командной строки. Имя каталога задаётся пользователем в качестве третьего аргумента командной строки. Программа выводит результаты поиска в файл (четвертый аргумент командной строки) в виде полный путь, имя файла, его размер. На консоль выводится общее число просмотренных файлов. Процедура поиска для каждого подкаталога должна запускаться в отдельном процессе. Каждый процесс выводит на экран свой ***pid,*** полный путь, имя и размер просмотренного файла, общее число просмотренных файлов в подкаталоге. Число запущенных процессов в любой момент времени не должно превышать ***N*** (вводится пользователем). Проверить работу программы для каталога***/usr/***размер ***31000 31500N=6***.
8. Написать программу нахождения массива значений функции ***y[i]=sin(2\*PI\*i/N) i= [0, N-1]***с использованием ряда Тейлора. Пользователь задаёт значения ***N***и количество ***n***членов ряда Тейлора. Для расчета каждого члена ряда Тейлора запускается отдельный процесс и его результат (член ряда) записывается в файл. Каждый процесс выводит на экран свой ***id*** и рассчитанное значение ряда. Головной процесс суммирует все члены ряда Тейлора, и полученное значение ***y[i]*** записывает в файл.Проверить работу программы для значений***N, n =[64,5]***и ***N, n =[32768,7]***.
9. Написать программу поиска одинаковых по их содержимому файлов в двух каталогов, например, ***Dir1*** и ***Dir2***. Пользователь задаёт имена ***Dir1*** и ***Dir2***. В результате работы программы файлы, имеющиеся в ***Dir1***, сравниваются с файлами в ***Dir2*** по их содержимому. Процедуры сравнения должны запускаться с использованием функции ***fork()*** в отдельном процессе для каждой пары сравниваемых файлов. Каждый процесс выводит на экран свой ***pid***, имя файла, число просмотренных байт и результаты сравнения. Число запущенных процессов любой момент времени не должно превышать ***N*** (вводится пользователем). Проверить работу программы для каталога***/usr/include/ и любого другого каталога в /home N=6***.
10. Написать программу поиска заданной пользователем строки из ***m*** байт (***m<255***) во всех файлах текущего каталога. Пользователь задаёт в качестве аргументов командной строки имя каталога, строку поиска, файл результата. Главный процесс открывает каталог и запускает для каждого файла каталога отдельный процесс поиска заданной комбинации из ***m*** байт. Каждый процесс выводит на экран и в свой ***pid***, полный путь и имя файла, число просмотренных в данном файле байт и результаты поиска (всё в одной строке!). Результаты поиска (только найденные файлы) по предыдущему формату записываются в выходной файл. Число запущенных процессов в любой момент времени не должно превышать ***N*** (вводится пользователем). Проверить работу программы для каталога ***/usr/include/ и строки “stdio.h”***
11. Разработать программу «интерпретатор команд», которая воспринимает команды, вводимые с клавиатуры, и осуществляет их корректное выполнение. Для этого каждая вводимая команда должна выполняться в отдельно запускаемом процессе с использованием вызова ***exec()***.Нельзя использовать вызов любого готового интерпретатора из своей программы или вызов ***system***().Для проверки работы, выполнить команду:***ls -l> 1.txt***. Предусмотреть контроль ошибок и команду выхода из программы.
12. То же что и в п.1, но вместо процессов использовать потоки.
13. То же что и в п.2, но вместо процессов использовать потоки.
14. То же что и в п.3, но вместо процессов использовать потоки.
15. То же что и в п.4, но вместо процессов использовать потоки.
16. То же что и в п.5, но вместо процессов использовать потоки.
17. То же что и в п.6, но вместо процессов использовать потоки.
18. То же что и в п.7, но вместо процессов использовать потоки.
19. То же что и в п.8, но вместо процессов использовать потоки.
20. То же что и в п.9, но вместо процессов использовать потоки.

## Контрольная работа №1

### Средства межпроцессного взаимодействия (сигналы)

**Цель работы – изучить методы и средства взаимодействия процессов с использованием сигналов в ОС Linux**

### Теоретическая часть

Все процессы в ***Linux*** выполняются в раздельных адресных пространствах и для организации межпроцессного взаимодействия необходимо использовать специальные средства:

1. общие файлы;
2. сигналы(***signal***);
3. каналы (***pipe***);
4. общую или разделяемую память;
5. семафоры.
6. При использовании общих файлов оба процесса открывают один и тот же файл, с помощью которого и обмениваются информацией. Для ускорения работы следует использовать файлы, отображаемые в памяти при помощи системного вызова ***mmap()***:

***#include <unistd.h>  
#include <sys/mman.h>***

***void \* mmap(void \*start, size\_t length, int prot , int flags, int fd, off\_t offset);***

Функция ***mmap*** отображает ***length*** байтов, начиная со смещения ***offset*** файла, определенного файловым описателем ***fd***, в память, начиная с адреса ***start***. Последний параметр ***offset*** необязателен, и обычно равен ***0***. Настоящее местоположение отраженных данных возвращается самой функцией ***mmap***, и никогда не бывает равным ***0***.Аргумент ***prot*** описывает желаемый режим защиты памяти (он не должен конфликтовать с режимом открытия файла):

***PROT\_EXEC*** данные в памяти могут исполняться;

***PROT\_READ*** данные в памяти можно читать;

***PROT\_WRITE***  в область можно записывать информацию;

***PROT\_NONE*** доступ к этой области памяти запрещен.

Параметр ***flags*** задает тип отражаемого объекта, опции отражения и указывает, принадлежат ли отраженные данные только этому процессу или их могут читать другие. Он состоит из комбинации следующих битов:

***MAP\_FIXED*** использование этой опции не рекомендуется;

***MAP\_SHARED разделить*** использование этого отражения с другими процессами, отражающими тот же объект. Запись информации в эту область памяти будет эквивалентна записи в файл. Файл может не обновляться до вызова функций ***msync*** или ***munmap***;

***MAP\_PRIVATE*** создать неразделяемое отражение с механизмом ***copy-on-write***. Запись в эту область памяти не влияет на файл. Не определено, являются или нет изменения в файле после вызова ***mmap*** видимыми в отраженном диапазоне.

1. Сигналы. С точки зрения пользователя получение процессом сигнала выглядит как возникновение прерывания. Процесс прерывает исполнение, и управление передается функции-обработчику сигнала. По окончании обработки сигнала процесс может возобновить регулярное исполнение. Типы сигналов принято задавать специальными символьными константами. Системный вызов ***kill()*** предназначен для передачи сигнала одному или нескольким специфицированным процессам в рамках полномочий пользователя.

***#include <sys/types.h>***

***#include <signal.h>***

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

Послать сигнал (не имея полномочий суперпользователя) можно только процессу, у которого эффективный идентификатор пользователя совпадает с эффективным идентификатором пользователя для процесса, посылающего сигнал. Аргумент ***pid*** указывает процесс, которому посылается сигнал, а аргумент ***sig*** – какой сигнал посылается. В зависимости от значения аргументов:

***pid > 0*** сигнал посылается процессу с идентификатором ***pid***;

***pid=0*** сигнал посылается всем процессам в группе, к которой принадлежит посылающий процесс;

***pid=-1*** и посылающий процесс не является процессом суперпользователя, то сигнал посылается всем процессам в системе, для которых идентификатор пользователя совпадает с эффективным идентификатором пользователя процесса, посылающего сигнал.

***pid = -1*** и посылающий процесс является процессом суперпользователя, то сигнал посылается всем процессам в системе, за исключением системных процессов (обычно всем, кроме процессов ***с pid = 0*** и ***pid = 1***).

***pid <0***, но не ***–1***, то сигнал посылается всем процессам из группы, идентификатор которой равен абсолютному значению аргумента ***pid*** (если позволяют привилегии).

если ***sig = 0***, то производится проверка на ошибку, а сигнал не посылается. Это можно использовать для проверки правильности аргумента ***pid*** (есть ли в системе процесс или группа процессов с соответствующим идентификатором).

Для того чтобы послать сигнал одновременно нескольким процессам их необходимо объединить в группу с помощью, например, функций **getpgrp()** или ***setpgid****()****.***

***int setpgrp(pid\_t pid, pid\_t pgid);***

***int setpgid(pid\_t pid, pid\_t pgid);***

Организация новой группы процессов выполняется системным вызовом ***getpgrp****(),* а получение собственного идентификатора группы процессов - системным вызовом ***getpgrp()*.**Функция ***setpgid****(*) присваивает идентификатор группы процессов ***pgid*** тому процессу, который был определен ***pid***. Если значение ***pid*** равно нулю, то процессу присваивается идентификатор текущего процесса. Если значение ***pgid***равно нулю, то используется идентификатор процесса, указанный ***pid***. Если ***setpgid*** используется для перевода процесса из одной группы в другую, то обе группы должны быть частью одной сессии. В этом случае ***pgid*** указывает на существующую группу процессов, с которой должен ассоциироваться процесс, а идентификатор сессии этой группы должен соответствовать идентификатору сессии присоединяющегося процесса. ***getpgid*** возвращает идентификатор группы процессов, к которой принадлежит процесс, указанный ***pid***. Если значение ***pid*** равно нулю, то используется идентификатор текущего процесса. Вызов ***setpgrp()***эквивалентен ***setpgid(0,0)****.*

Аналогично, значение ***getpgrp()*** эквивалентно ***getpgid(0)****.*

Системные вызовы для установки собственного обработчика сигналов:

***#include<signal.h>***

***void (\*signal (intsig, void (\*handler) (int)))(int);***

***int sigaction(int sig, const struct sigaction \*act,struct sigaction \*oldact);***

Структура ***sigaction*** имеет следующий формат:

***struct sigaction {***

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

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

***sigset\_t sa\_mask;***

***int sa\_flags;***

***void (\*sa\_restorer)(void);***

Системный вызов ***signal*** служит для установки обработчика сигнала для процесса. Параметр ***sig*** – это номер сигнала, обработку которого предстоит изменить. Параметр ***handler*** описывает новый способ обработки сигнала – это может быть указатель на пользовательскую функцию-обработчик сигнала, специальное значение ***SIG\_DFL*** (восстановить реакцию процесса на сигнал ***sig*** по умолчанию) или специальное значение ***SIG\_IGN*** (игнорировать поступивший сигнал ***sig***). Системный вызов возвращает указатель на старый способ обработки сигнала, значение которого можно использовать для восстановления старого способа в случае необходимости.

Пример пользовательской обработки сигнала ***SIGUSR1***.

***void \*my\_handler(intnsig) {*** кодфункции-обработчикасигнала ***}***

***int main() {***

***(void) signal(SIGUSR1, my\_handler); }***

Системный вызов ***sigaction*** используется для изменения действий процесса при получении соответствующего сигнала. Параметр ***sig*** задает номер сигнала и может быть равен любому номеру. Если параметр ***act*** не равен нулю, то новое действие, связанное с сигналом ***sig***, устанавливается соответственно ***act***. Если ***oldact*** не равен нулю, то предыдущее действие записывается в ***oldact***.

### Варианты индивидуальных заданий

Создать дерево процессов согласно варианта индивидуального задания. Номер варианта индивидуального задания

Дерево процессов = (Номер Вашего паспорта) mod 15.

Последовательность обмена сигналами = (Номер Вашего паспорта) mod 13.

Процессы непрерывно обмениваются сигналами согласно табл. 2. Запись в таблице 1 вида: 1***-> (2,3,4,5)*** означает, что исходный процесс ***0*** создаёт дочерний процесс ***1,*** который, в свою очередь, создаёт дочерние процессы ***2,3,4,5.*** Запись в таблице 2 вида: 1***-> (2,3,4) SIGUSR1*** означает, что процесс 1 посылает дочерним процессам 2***,3,4*** одновременно (т.е. за один вызов kill ()) сигнал ***SIGUSR1***. После передачи ***101***–го по счету сигнала ***SIGUSR***родительский процесс посылает сыновьям сигнал ***SIGTERM***и ожидает завершения всех сыновей, после чего завершается***.*** Сыновья, получив сигнал ***SIGTERM*** завершают работу с выводом на консоль сообщения вида:

***Pid ppid завершил работу после X-го сигналаSIGUSR1 и Y-го сигналаSIGUSR2***

***где X, Y – количество посланных за все время работы данным сыном сигналов SIGUSR1 и SIGUSR2***

Каждый процесс в процессе работы выводит на консоль информацию в следующем виде:

***Npidppid послал/получил USR1/USR2 текущее время (мксек)***

***где N-номер сына по табл. 1***

Варианты индивидуальных заданий в табл.1, табл.2.

**Таблица 1. Дерево процессов**

|  |  |
| --- | --- |
| **№** | **Дерево процессов** |
| **0** | **1->2 2->(3,4) 4->5 3->6 6->7 7->8** |
| **1** | **1->(2,3,4) 2->(5,6) 6->7 7->8** |
| **2** | **1->(2,3,4,5) 2->6 3->7 4->8** |
| **3** | **1->(2,3) 2->(4,5) 5->6 6->(7,8)** |
| **4** | **1->(2,3,4,5) 5->(6,7,8)** |
| **5** | **1->(2,3) 3->4 4->(5,6,7) 7->8** |
| **6** | **1->2 2->(3,4) 4->5 3->6 6->7 7->8** |
| **7** | **1->(2,3,4,5,6) 6->(7,8)** |
| **8** | **1->2 2->(3,4,5) 4->6 3->7 5->8** |
| **9** | **1->2 2->3 3->(4,5,6)6->7 4->8** |
| **10** | **1->(2,3) 3->4 4->(5,6) 6-7 7->8** |
| **11** | **1->2 2->(3,4) 4->5 3->6 6->7 7->8** |
| **12** | **1->(2,3,4,5,6,7) 2,3,4,5,6,7->8** |
| **13** | **1->2 2->(3,4,5) 4->6 3->7 5->8** |
| **14** | **1->2 2->3 3->(4,5,6)6->7 4->8** |
| **15** | **1->(2,3,4,5) 2->(6,7) 7->8** |

**Таблица 2. Последовательность обмена сигналами**

|  |  |
| --- | --- |
| **№** | **Последовательность обмена сигналами** |
| **0** | ***1->2SIGUSR1 2->(3,4)SIGUSR2 4->5 SIGUSR1***  ***3->6SIGUSR1 6->7SIGUSR1 7->8SIGUSR2 8->1SIGUSR2*** |
| **1** | ***1->(2,3,4)SIGUSR1 2->(5,6)SIGUSR2 6->7SIGUSR1***  ***7->8SIGUSR1 8->1SIGUSR2*** |
| **2** | ***1->(2,3,4,5) SIGUSR22->6SIGUSR1 3->7SIGUSR1 4->8SIGUSR1 8->1SIGUSR1*** |
| **3** | ***1->(2,3) SIGUSR1 2->(4,5)SIGUSR1 5->6SIGUSR1***  ***6->(7,8)SIGUSR1 8->1SIGUSR1*** |
| **4** | ***1->(1,2,3,4,5)SIGUSR1 5->(6,7,8)SIGUSR18->1SIGUSR1*** |
| **5** | ***1->(2,3)SIGUSR1 3->4SIGUSR2 4->(5,6,7)SIGUSR1***  ***7->8SIGUSR1 8->1SIGUSR2*** |
| **6** | ***1->2SIGUSR1 2->(3,4)SIGUSR2 4->5SIGUSR1***  ***3->6SIGUSR1 6->7SIGUSR1 7->8SIGUSR1 8->1SIGUSR1*** |
| **7** | ***1->(2,3,4,5,6)SIGUSR2 6->(7,8)SIGUSR18->1SIGUSR2*** |
| **8** | ***1->2SIGUSR2 2->(3,4,5)SIGUSR1 4->6SIGUSR1***  ***3->7SIGUSR1 5->8 SIGUSR1 8->1SIGUSR2*** |
| **9** | **1->(8,7,6)*SIGUSR1*8->4*SIGUSR1*7->4*SIGUSR2***  **6->4*SIGUSR1*4->(3,2)*SIGUSR1*2->1*SIGUSR2*** |
| **10** | **1->(8,7)*SIGUSR1*8->(6,5)*SIGUSR1*5->(4,3,2)*SIGUSR2***  **2->1*SIGUSR2*** |
| **11** | **1->(8,7,6,5)*SIGUSR1*8->3*SIGUSR1*7->3*SIGUSR2***  **6->3*SIGUSR1*5->3 *SIGUSR1* 3->2*SIGUSR2*2->1 *SIGUSR2*** |
| **12** | **1->6*SIGUSR1*6->7*SIGUSR1*7->(4,5)*SIGUSR2***  **4->8*SIGUSR1*5->2*SIGUSR1* 8->2*SIGUSR2*2->1*SIGUSR2*** |
| **13** | **1->8*SIGUSR1*8->7*SIGUSR1*7->(4,5,6)*SIGUSR2***  **4->2*SIGUSR1*2->3 *SIGUSR1* 3->1*SIGUSR2*** |

## Контрольная работа №2

### Средства межпроцессного взаимодействия (каналы, разделяемая память, семафоры)

**Цель работы – изучить методы и средства взаимодействия процессов с использованием каналов, разделяемой памяти и семафоров в ОС Linux.**

### *Варианты индивидуальных заданий*

Номер варианта индивидуального задания K = (Номер Вашего паспорта) mod 5.

1. Cоздать два дочерних процесса. Родительский процесс создаёт семафор (***сем1***) и ***2*** неименованных канала (***кан1 и кан2)***. Оба дочерних процесса непрерывно записывают в каналы по ***110*** строк вида: ***номер\_строки pid\_процесса текущее\_время*** (мксек). Родительский процесс читает из каждого канала по ***75*** строк и выводит их на экран. Всего дочерние процессы должны записать по ***1010*** строк. Семафор (***сем1***) используется процессами для разрешения кому из процессов получить доступ к каналу. Дочерние процессы начинают работу после получения сигнала ***SIGUSR2*** от родительского процесса. По завершению работы они посылают сигнал ***SIGUSR1***родительскому процессу.
2. Cоздать два дочерних процесса. Родительский процесс создаёт семафор (***сем1***) и разделяемую память. Оба дочерних процесса непрерывно записывают в разделяемую память по ***75*** строк вида: ***номер\_строки pid\_процесса текущее\_время*** (мксек). Всего процессы должны записать ***1000*** строк. Семафор ***сем1*** используется процессами для разрешения кому из процессов получить доступ к разделяемой памяти. Родительский процесс читает из разделяемой памяти по ***75*** строк и выводит их на экран. Дочерние процессы начинают работу после получения сигнала ***SIGUSR2*** от родительского процесса. По завершению работы они посылают сигнал ***SIGUSR1***родительскому процессу.
3. Cоздать два дочерних процесса. Родительский процесс создаёт семафор (***сем1***) и общий файл. Дочерние процессы записывают в файл по 3 строки за раз всего ***1100*** строк вида: ***номер\_строки pid\_процесса текущее\_время*** (мксек). Родительский процесс читает из файла по 3 строки и выводит их на экран в следующем виде: ***pid строка\_прочитанная\_из\_файла***. Семафор ***сем1*** используется процессами для разрешения, кому из процессов получить доступ к файлу. Дочерние процессы начинают работу после получения сигнала ***SIGUSR2*** от родительского процесса. По завершению работы они посылают сигнал ***SIGUSR1***родительскому процессу.
4. Написать программу, создающую 2 дочерних процесса. Родительский процесс создаёт 2 неименованных канала. Дочерние процессы записывают в канал по 100 строк вида: ***номер\_строкиpid\_процессатекущее\_время*** (мксек). Родительский процесс читает из канала по 75 строк и выводит их на экран в следующем виде: ***pid строка, прочитанная из файла***. Дочерние процессы начинают работу после получения сигнала ***SIGUSR2*** от родительского процесса. По завершению работы они посылают сигнал ***SIGUSR1***родительскому процессу.
5. Cоздать два дочерних процесса. Родительский процесс создаёт семафор (***сем1***) и общий файл отображенный в память. Оба дочерних процесса непрерывно записывают в файл по ***100*** строк вида: ***номер\_строки pid\_процесса текущее\_время*** (мсек). Всего процессы должны записать ***1000*** строк. Семафор ***сем1*** используется процессами для разрешения кому из процессов получить доступ к файлу. Родительский процесс читает из файла по ***75*** строк и выводит их на экран. Дочерние процессы начинают работу после получения сигнала ***SIGUSR2*** от родительского процесса. По завершению работы они посылают сигнал ***SIGUSR1***родительскому процессу.

### Теоретическая часть

Каналы. Программный канал – это файл особого типа (***FIFO***: «первым вошел – первым вышел»). Процессы могут записывать и считывать данные из канала как из обычного файла. Если канал заполнен, процесс записи в канал останавливается до тех пор, пока не появится свободное место, чтобы снова заполнить его данными. С другой стороны, если канал пуст, то читающий процесс останавливается до тех пор, пока пишущий процесс не запишет данные в этот канал. В отличие от обычного файла здесь нет возможности позиционирования по файлу с использованием указателя.

В ОС Linux различают два вида программных каналов:

* Именованный программный канал. Именованный программный канал может служить для общения и синхронизации произвольных процессов, знающих имя данного программного канала и имеющих соответствующие права доступа. Для создания используется вызов:

***int mkfifo(const char \*filename, mode\_t mode);***

* Неименованный программный канал. Неименованным программным каналом могут пользоваться только создавший его процесс и его потомки. Для создания используется вызов:

***int pipe(int fd[2]);***

Использование разделяемой памяти заключается в создании специальной области памяти, позволяющей иметь к ней доступ нескольким процессам. Системные вызовы для работы с разделяемой памятью:

***intshmget(key\_tkey, intsize, intshmflg);***

***int shm\_open (const char \*name, int oflag, mode\_t mode);***

Системный вызов***shmget*** предназначен для выполнения операции доступа к сегменту разделяемой памяти и, в случае ее успешного завершения, возвращает дескриптор ***System V IPC*** для этого сегмента (целое неотрицательное число, однозначно характеризующее сегмент внутри вычислительной системы и использующееся в дальнейшем для других операций с ним). Параметр ***key*** является ключом ***System V IPC*** для сегмента, т.е. фактически его именем из пространства имен ***System V IPC***. В качестве значения этого параметра может быть использовано значение ключа, полученное с помощью функции [***ftok()***](http://unilab.math.spbu.ru/archive/OS/man/ftok.htm), или специальное значение***IPC\_PRIVATE****.* Использование значения***IPC\_PRIVATE*всегда** приводит к попытке создания нового сегмента разделяемой памяти с ключом, который не совпадает со значением ключа ни одного из уже существующих сегментов и который не может быть получен с помощью функции [***ftok()***](http://unilab.math.spbu.ru/archive/OS/man/ftok.htm) ни при одной комбинации ее параметров. Параметр ***size*** определяет размер создаваемого или уже существующего сегмента в байтах. В случае если сегмент с указанным ключом уже существует, но его размер не совпадает с указанным в параметре ***size***, констатируется возникновение ошибки.  
Параметр ***shmflg*** - флаги - играет роль только при создании нового сегмента разделяемой памяти и определяет права различных пользователей при доступе к сегменту, а также необходимость создания нового сегмента и поведение системного вызова при попытке создания. Он является некоторой комбинацией (с помощью операции побитовое или - "|") следующих предопределенных значений и восьмеричных прав доступа:

|  |  |
| --- | --- |
| ***IPC\_CREAT*** | - если сегмент для указанного ключа не существует, он должен быть создан. |
| ***IPC\_EXCL*** | - применяется совместно с флагом ***IPC\_CREAT***. При совместном их использовании и существовании сегмента с указанным ключом, доступ к сегменту не производится и констатируется ошибочная ситуация, при этом переменная ***errno***, описанная в файле ***errno.h***, примет значение ***EEXIST***. |
| **0400** | - Разрешено чтение для пользователя, создавшего сегмент. |
| **0200** | - Разрешена запись для пользователя, создавшего сегмент. |
| ***0040*** | - Разрешено чтение для группы пользователя, создавшего сегмент. |
| ***0020*** | - Разрешена запись для группы пользователя, создавшего сегмент. |
| ***0004*** | - Разрешено чтение для всех остальных пользователей |
| ***0002*** | - Разрешена запись для всех остальных пользователей |

***#include <sys/mman.h>***

***int shm\_open (const char \*name, int oflag, mode\_t mode);***

***int shm\_unlink (const char \*name);***

Вызов ***shm\_open*** создает и открывает новый (или уже существующий) объект разделяемой памяти. При открытии с помощью функции ***shm\_open()*** возвращается файловый дескриптор. Имя ***name*** трактуется стандартным для рассматриваемых средств межпроцессного взаимодействия образом. Посредством аргумента ***oflag*** могут указываться флаги ***O\_RDONLY, O\_RDWR, O\_CREAT, O\_EXCL*** и/или ***O\_TRUNC***. Если объект создается, то режим доступа к нему формируется в соответствии со значением ***mode*** и маской создания файлов процесса. Функция ***shm\_unlink*** выполняет обратную операцию, удаляя объект, предварительно созданный с помощью ***shm\_open***. После подключения сегмента разделяемой памяти к виртуальной памяти процесса этот процесс может обращаться к соответствующим элементам памяти с использованием обычных машинных команд чтения и записи, не прибегая к использованию дополнительных системных вызовов.

***int main (void) {***

***int fd\_shm;*** /\* Дескриптор объекта в разделяемой памяти\*/

***if ((fd\_shm = shm\_open (“myshered.shm”, O\_RDWR | O\_CREAT, 0777)) < 0) { perror ("error create shm"); return (1); }***

Для компиляции программы необходимо подключить библиотеку ***rt.lib*** следующим способом: ***gcc 1.c –o 1.exe -lrt***

1. Семафор – переменная определенного типа, которая доступна параллельным процессам для проведения над ней только двух операций:

* ***A(S, n)*** – увеличить значение семафора ***S*** на величину ***n***;
* ***D(S, n)*** – если значение семафора ***S < n***, процесс блокируется. Далее ***S = S - n***;
* ***Z(S)*** – процесс блокируется до тех пор, пока значение семафора ***S*** не станет равным 0.

Семафор играет роль вспомогательного критического ресурса, так как операции ***A*** и ***D*** неделимы при своем выполнении и взаимно исключают друг друга. Семафорный механизм работает по схеме, в которой сначала исследуется состояние критического ресурса, а затем уже осуществляется допуск к критическому ресурсу или отказ от него на некоторое время. Основным достоинством семафорных операций является отсутствие состояния «активного ожидания», что может существенно повысить эффективность работы мультипрограммной вычислительной системы.

Для работы с семафорами имеются следующие системные вызовы:

Создание и получение доступа к набору семафоров:

***int semget(key\_t key, int nsems, int semflg);***

Параметр ***ke***y является ключом для массива семафоров, т.е. фактически его именем. В качестве значения этого параметра может использоваться значение ключа, полученное с помощью функции ***ftok()***, или специальное значение ***IPC\_PRIVATE***. Использование значения ***IPC\_PRIVATE*** всегда приводит к попытке создания нового массива семафоров с ключом, который не совпадает со значением ключа ни одного из уже существующих массивов и не может быть получен с помощью функции ***ftok()*** ни при одной комбинации ее параметров. Параметр ***nsems*** определяет количество семафоров в создаваемом или уже существующем массиве. В случае если массив с указанным ключом уже имеется, но его размер не совпадает с указанным в параметре ***nsems***, констатируется возникновение ошибки.

Параметр ***semflg*** – флаги – играет роль только при создании нового массива семафоров и определяет права различных пользователей при доступе к массиву, а также необходимость создания нового массива и поведение системного вызова при попытке создания. Он является некоторой комбинацией (с помощью операции побитовое или – "**|**") следующих предопределенных значений и восьмеричных прав доступа:

***IPC\_CREAT*** — если массива для указанного ключа не существует, он должен быть создан;

***IPC\_EXCL*** — применяется совместно с флагом ***IPC\_CREAT***. При совместном их использовании и существовании массива с указанным ключом, доступ к массиву не производится и констатируется ошибка, при этом переменная ***errno***, описанная в файле <***errno.h***>, примет значение ***EEXIST***;

***0400*** — разрешено чтение для пользователя, создавшего массив

***0200*** — разрешена запись для пользователя, создавшего массив

***0040*** — разрешено чтение для группы пользователя, создавшего массив

***0020***— разрешена запись для группы пользователя, создавшего массив

***0004***— разрешено чтение для всех остальных пользователей

***0002***— разрешена запись для всех остальных пользователей

Пример: semflg***= IPC\_CREAT*|*0022***

Изменение значений семафоров:

***int semop(int semid, struct sembuf \*sops, int nsops);***

Параметр ***semid*** является дескриптором System V IPC для набора семафоров, т. е. значением, которое вернул системный вызов ***semget ()*** при создании набора семафоров или при его поиске по ключу. Каждый из ***nsops*** элементов массива, на который указывает параметр ***sops***, определяет операцию, которая должна быть совершена над каким-либо семафором из массива IPC семафоров, и имеет тип структуры:

***struct sembuf {***

***short sem\_num;*** //номер семафора в массиве IPC семафоров (начиная с ***0***);

***short sem\_op;***//выполняемая операция;

***short sem\_flg***; // флаги для выполнения операции.

***}***

Значение элемента структуры ***sem\_op*** определяется следующим образом:

* + для выполнения операции **A(S,n)**значение должно быть равно ***n***;
  + для выполнения операции ***D(S,n)*** значение должно быть равно -***n***;
  + для выполнения операции ***Z(S)*** значение должно быть равно ***0***.

Семантика системного вызова подразумевает, что все операции будут в реальности выполнены над семафорами только перед успешным возвращением из системного вызова. Если при выполнении операций ***D*** или ***Z*** процесс перешел в состояние ожидания, то он может быть выведен из этого состояния при возникновении следующих форс-мажорных ситуаций: массив семафоров был удален из системы; процесс получил сигнал, который должен быть обработан.

Выполнение разнообразных управляющих операций (включая удаление) над набором семафоров:

***intsemctl(intsemid, intsemnum, intcmd, unionsemunarg);***

Изначально все семафоры инициируются нулевым значением.

# Контроль знаний

## Вопросы к экзамену

* + - 1. Основные принципы построения ОС UNIX.
      2. Состав ядра ОС UNIX.
      3. Файлы и файловая система в ОС UNIX.
      4. Атрибуты файла в ОС UNIX.
      5. Полное имя файла, каталог файлов.
      6. Функции создания файлов, открытия файлов. Обработка ошибок.

1. Функции записи данных в файл, чтения данных из файла. Обработка ошибок.
2. Категории владельцев файла. Права доступа к файлам.
3. Разработка программ в ОС UNIX. Обработка ошибок.
4. Процессы в ОС UNIX. Типы процессов.
5. Атрибуты процесса в ОС UNIX.
6. Системный вызов порождения процесса.
7. Системные вызовы перегрузки процесса.
8. Системный вызов завершения процесса.
9. Системные вызовы ожидания завершения процесса.
10. Типы сигналов в ОС UNIX. Их назначение.
11. Способы обработки сигналов.
12. Функция signal, ее назначение и применение.
13. Функция sigaction, ее назначение и применение.
14. Маска сигналов, системные вызовы посылки сигналов.
15. Каналы ОС UNIX. Системный вызов pipe.
16. Взаимодействие процессов посредством каналов.
17. Ограничения в ОС UNIX.
18. Очереди сообщений.
19. Взаимодействие процессов с применением семафоров.
20. Разделяемая память.
21. Понятие потока в ОС UNIX.
22. Понятие операционной системы. Основные этапы развития ОС.
23. Функции и свойства операционных систем.
24. Особенности современных операционных систем.
25. Концепция слоистой операционной системы и системы на основе микроядра.
26. Функции микроядра.
27. Принципы построения ОС.
28. Понятие процесса, модели процессов.
29. Описание процесса.
30. Концепция потока, как составной части процесса.
31. Концепция виртуализации.
32. Дисциплины распределения ресурсов.
33. Подсистема управления памятью. Требования, предъявляемые к ней.
34. Распределение памяти. Система двойников.
35. Виртуальная память. Задачи управления виртуальной памятью.
36. Задача замещения при управлению виртуальной памятью, часовой алгоритм.
37. Модели организации ввода-вывода.
38. Взаимодействие последовательных процессов. Задача взаимного исключения. Вариант 1.
39. Взаимодействие последовательных процессов. Задача взаимного исключения. Вариант 2.
40. Взаимодействие последовательных процессов. Задача взаимного исключения. Вариант 3.
41. Взаимодействие последовательных процессов. Задача взаимного исключения. Вариант 4.
42. Взаимодействие последовательных процессов. Задача взаимного исключения. Вариант 5.
43. Задача взаимного исключения. Алгоритм Петерсона.
44. Синхронизирующие примитивы. Решение задачи взаимного исключения с использованием семафоров.
45. Применение общего семафора для решения задачи "производитель-потребитель" с неограниченным буфером.
46. Применение двоичных семафоров для решения задачи "производитель-потребитель" (буфер неограниченный).
47. Применение семафоров для решения задачи "производитель-потребитель" с неограниченным буфером. Решение "спящий парикмахер".
48. Применение общих семафоров для решения задачи "производитель-потребитель" с ограниченным буфером.
49. Взаимодействие через переменные состояния. Пример приоритетного правила.
50. Проблема тупиков. Алгоритм банкира.
51. Архитектура процессора IA-32 , как типовая архитектура многозадачного режима.
52. Организация внутренней кэш-памяти процессора Intel архитектуры.
53. Устройство страничного преобразования IA-32. Организация ассоциативного буфера преобразования.
54. Кэширование памяти. Кэш прямого отображения. Полностью ассоциативный кэш.
55. Сегментация памяти в процессорах IA-32.
56. Формат дескриптора сегмента при работе в защищенном режиме.
57. Дескрипторные таблицы. Селекторы сегментов. Дескриптор ЛДТР.
58. Последовательность действий по преобразованию адреса в защищенном режиме.
59. Структура элементов каталога страниц и таблиц страниц.
60. Страничная организация памяти. Схема преобразования линейного адреса в физический.
61. Организация защиты при работе в защищенном режиме. Определение уровней привилегий.
62. Защита доступа к данным, поле PRL
63. Защита сегмента кода.
64. Страничная организация памяти. Схема преобразования линейного адреса в физический.
65. Организация защиты при работе в защищенном режиме. Определение уровней привилегий.
66. Передача управления между различными уровнями привилегий, подчиненные сегменты.
67. Передача управления между различными уровнями привилегий, шлюзы вызова.
68. Поддержка многозадачности процессорами Intel.
69. Переключение задач в многозадачном режиме.
70. Регистр задачи. Вложенность задач.
71. Прерывания и особые случаи в процессоре 8086.
72. Прерывания и особые случаи при работе процессора в защищенном режиме.

## Тесты

ВОПРОСЫ ТЕСТА 1

1. Какие поля содержит запись о пользователе в файле регистрации пользователей?

2. Какое значение возвращают системные вызовы при завершении работы с ошибкой?

3. Где помещается код ошибки при ошибочном завершении системного вызова?

4. Какие типы файлов существуют в ОС UNIX?

5. Какие классы (категории) пользователей существуют при определение прав доступа к файлу?

6. Какие типы прав доступа к файлу устанавливаются для каждого класса пользователей?

7. Кто может изменить права доступа к файлу?

8. Какая команда используется для изменения прав доступа?

9. Какие основные системные функции используются при работе с файлами?

10. Где хранятся характеристики файла?

11. Какие системные вызовы применяются при чтении метаданных файла?

12. Из каких основных частей состоит файловая система UNIX?

ВОПРОСЫ ТЕСТА 2

1. Какие типы процессов имеют место в UNIX?

2. Какие основные атрибуты процесса?

3. Какой системный вызов используется для создания (порождения) нового процесса?

4. Какой системный вызов используется для определения идентификатора текущего процесса?

5. Какой системный вызов используется для определения идентификатора родительского процесса?

6. Для чего применяются системные вызовы типа exec?

7. Какое системные вызовы используются для ожидания завершения порожденного процесса?

8. Какие средства можно использовать для организации взаимодействия между процессами?

9. Какой системный вызов используется для создания канала?

10. Какой системный вызов используется для посылки сигнала процессу или группе процессов?

11. Какие типы действий могу устанавливаться для выполнения при получении сигнала?

12. Какие системные вызовы применяются для установки обработчиков на сигнал?

13. Какой системный вызов используется для получения установленных ограничений для процесса?

ВОПРОСЫ ТЕСТА 3

1. Определение процесса?

2. В каких состояниях может находиться процесс

3. Определение ресурса?

4. Что такое виртуальный ресурс?

5. Сущность задачи взаимного исключения.

6. Какие положения следует проверить для доказательства решения задачи взаимного исключения?

7. Какие операции выполняются над семафорами?

8. В чем сущность V - операции?

9. В чем сущность P - операции?

10. Как выглядит решение задачи взаимного исключения с помощью семафоров?

11. В чем сущность задачи производитель-потребитель?

12. Какая ситуация называется тупиком?

13. На чем основана концепция монитороподобных средств?

14. Что означает описание переменной V: SHARED T ?

15. Для чего используется конструкция вида REGION V DO S ?

### Ответы на тесты

ВОПРОСЫ ТЕСТА 1

1. Какие поля содержит запись о пользователе в файле регистрации пользователей?

(регистрационное имя, пароль, идентификатор пользователя, идентификатор первичной группы пользователя, расширенная информация о пользователе, домашний каталог пользователя, имя командного интерпретатора)

2. Какое значение возвращают системные вызовы при завершении работы с ошибкой? (-1)

3. Где помещается код ошибки при ошибочном завершении системного вызова?

( во внешней переменной errno)

4. Какие типы файлов существуют в ОС UNIX?

( Обычный файл, Каталог, Специальный файл устройства,

Файл FIFO или именованный канал, Связь (link), Сокет (socket)

5. Какие классы (категории) пользователей существуют при определение прав доступа к файлу?

(Владелец файла, Группа-владелец файла, Остальные пользователи)

6. Какие типы прав доступа к файлу устанавливаются для каждого класса пользователей?

(На чтение, на запись, на выполнение)

7. Кто может изменить права доступа к файлу?

(Владелец файла, Суперпользователь)

8. Какая команда используется для изменения прав доступа?

(chmod)

9. Какие основные системные функции используются при работе с файлами?

(creat, open, read, write, close, lseek, dup,fcntl)

10. Где хранятся характеристики файла?

(в индексном дескрипторе)

11. Какие системные вызовы применяются при чтении метаданных файла?

(lstat, fstat)

12. Из каких основных частей состоит файловая система UNIX?

(суперблок, массив индексных дескрипторов, блоки хранения данных)

ВОПРОСЫ ТЕСТА 2

1. Какие типы процессов имеют место в UNIX?

(системные процессы, прикладные процессы, процессы-демоны)

2. Какие основные атрибуты процесса?

(Идентификатор процесса, Идентификатор родительского процесса, Реальный идентификатор пользователя, Эффективный идентификатор пользователя, Реальный идентификатор группы, Эффективный идентификатор группы, Приоритет процесса, Терминальная линия)

3. Какой системный вызов используется для создания (порождения) нового процесса?

(fork)

4. Какой системный вызов используется для определения идентификатора текущего процесса?

(getpid)

5. Какой системный вызов используется для определения идентификатора родительского процесса?

(getppid)

6. Для чего применяются системные вызовы типа exec?

(Для перегрузки (перезапуска) текущей программы)

7. Какое системные вызовы используются для ожидания завершения порожденного процесса?

(wait, waitpid)

8. Какие средства можно использовать для организации взаимодействия между процессами?

(файлы, каналы, сигналы)

9. Какой системный вызов используется для создания канала?

(pipe)

10. Какой системный вызов используется для посылки сигнала процессу или группе процессов?

(kill)

11.Какие типы действий могу устанавливаться для выполнения при получении сигнала?

(игнорировать сигнал, выполнение действий по умолчанию, перехват и самостоятельная обработка сигнала)

12. Какие системные вызовы применяются для установки обработчиков на сигнал?

(signal, sigaction)

13. Какой системный вызов используется для получения установленных ограничений для процесса?

(getrlimit)

ВОПРОСЫ ТЕСТА 3

1. Определение процесса? (процесс - система действий, реализующая определенную функцию в вычислительной системе и оформленная так, что управляющая программа вычислительной системы может перераспределять ресурсы этой системы в целях обеспечения мультипрограммирования)

2. В каких состояниях может находиться процесс(порождение, готовность, активное, ожидание, окончание)

3. Определение ресурса?(ресурсом является средство вычислительной системы, которое может быть выделено процессу на определенный интервал времени)

4. Что такое виртуальный ресурс? (Виртуальный ресурс - это некоторая модель физического ресурса. Виртуальный ресурс не существует в том виде, в котором он проявляет себя пользователю. Как модель виртуальный ресурс реализуется в некоторой программно-аппаратной форме. В этом смысле виртуальный ресурс существует. Однако виртуальный ресурс может предоставить пользователю не только часть тех свойств, которые присущи объекту моделирования, т.е. физическому ресурсу, но и свойства, которые ему не присущи. Однако пользователь будет воспринимать их, как реально существующие.)

5. Сущность задачи взаимного исключения.(рассматривается два последовательных процесса, которые удобно считать циклическими. В каждом цикле выполнения процесса существует критический интервал. Это означает, что в любой момент времени только один процесс может находиться внутри своего критического интервала или осуществляется их взаимное исключение)

6. Какие положения следует проверить для доказательства решения задачи взаимного исключения?

* Решение безопасно в том смысле, что два процесса не могут одновременно оказаться в своих критических интервалах.
* В случае сомнения, кому из двух процессов первому войти в критический интервал, выяснение этого вопроса не откладывается до бесконечности.
* Остановка какого-либо из процессов в остатке цикла не вызывает блокировки другого процесса.

7. Какие операции выполняются над семафорами?

(P - операция и V - операция )

8. В чем сущность V - операции?

(V-операция – операция с одним аргументом, который должен быть семафором. Ее назначение – увеличение аргумента на единицу. Это действие рассматривается как неделимая операция. )

9. В чем сущность P - операции?

(p-операция – операция с одним аргументом, который должен быть семафором. Ее назначение – уменьшение аргумента на единицу, если только результирующее значение не становится отрицательным. Завершение р-операции, то есть решение о том, что в настоящий момент является подходящим для выполнения уменьшения и последующее уменьшение значения аргумента, рассматривается как неделимая операция.

р-операция определяет потенциальную задержку. Если инициируется р-операция над семафором, который в этот момент равен нулю, то в данном случае р-операция не может завершиться, пока какой-либо другой процесс не выполнит v-операцию над тем же семафором и не присвоит ему значение 1.)

10. Как выглядит решение задачи взаимного исключения с помощью семафоров?

(процесс i:

begin

Li: p(свободно);

Критический интервал i;

V(свободно);

Остатокцикла i;

goto Li;

end; )

11. В чем сущность задачи производитель-потребитель?

(Рассматриваются два процесса, которые называются производитель и потребитель. Оба процесса являются циклическими. Производитель при каждом циклическом повторении участка программы производит отдельную порцию информации, которая должна быть обработана потребителем. Потребитель при каждом повторении обрабатывает следующую порцию информации, вырабо-танную производителем. Отношения производитель-потребитель подразумевают односторонний канал связи, по которому могут передаваться порции информации. С этой целью процессы связаны через буфер неограниченной емкости. То есть, произведенные порции не должны немедленно потребляться, а могут организовывать в буфере очередь. Буфер работает по принципу FIFO.)

12. Какая ситуация называется тупиком?

(Ситуация, когда какой-либо из процессов может быть завершён лишь при условии уничтожения какого-либо другого процесса, называется «смертельными объятиями» или тупиком.)

13. На чем основана концепция монитороподобных средств?

(Концепция монитороподобных средств основана на языковом обособлении или локализации средств взаимодействия. Это не только выделение особых языковых конструкций, но и сосредоточение в их составе информации о разделяемых ресурсах, о переменных состояния, характеризующих эти ресурсы, а также допустимые действия над этими ресурсами.)

14. Что означает описание переменной V: SHARED T ?

(Описание переменной V: SHARED T означает, что определяется ресурс с именем V, как некоторая переменная, доступная параллельным процессам. Тип ресурса задаётся его описанием T. )

15. Для чего используется конструкция вида REGION V DO S ?

(Для осуществления доступа к ресурсу V в тексте программы процесса требуется использовать конструкцию вида REGION V DO S. Такая конструкция описывает отдельную критическую область относительно критического ресурса V и определяет действия S, которые будут осуществлены над ресурсом. Такие конструкции при исполнении исключают друг друга относительно критического ресурса.)

### Вопросы для самоконтроля по второй части

* + - 1. В чем состоит принцип адресации в ассоциативной памяти.
      2. Какие системные регистры есть у процессора IA32.
      3. Где хранится адрес каталога страниц.
      4. Какие основные поля содержит дескриптор сегмента данных (кода).
      5. Какие есть типы дескрипторных таблиц.
      6. Какая структура селектора сегментов при работе в защищенном режиме.
      7. Где хранится селектор локальной дескрипторной таблицы.
      8. Как организуется преобразование логического адреса в линейный при работе процессора в защищенном режиме.
      9. Как организуется преобразование линейного адреса в физический при поддержке страничной организации памяти.
      10. Какие достоинства имеет страничная организация памяти.
      11. Какие у процессора есть привилегированные команды.
      12. Какие существуют принципы организации защиты при работе процессора в защищенном режиме.
      13. Где хранится текущий уровень привилегий.
      14. Для чего используются подчиненные сегменты.
      15. Для чего используются шлюзы вызова.
      16. Как контролируются права при доступе к шлюзу вызова.
      17. Что обозначает запрашиваемый уровень привилегий.
      18. Какие средства имеет процессор IA32 для поддержки многозадачности.
      19. Какие команды инициируют переключение задач.
      20. Как организован возврат к предыдущей задаче при вложенности задач.
      21. Какие дескрипторы может содержать дескрипторная таблица прерываний.
      22. Определение операционной системы.
      23. Принципы построения ОС.
      24. Какие услуги предоставляет операционная система.
      25. Какие характерные особенности у современных операционных систем.
      26. Какие основные функции ОС на основе микроядра.
      27. Какие достоинства ОС на основе микроядра.
      28. Какая информация необходима для описания процесса.
      29. В чем смысл многопоточности.
      30. Какие применяются модели процессов.
      31. Какие есть формы виртуализации.
      32. Что представляет собой идеальная виртуальная машина.
      33. Какие есть схемы распределения оперативной памяти.
      34. Какие задачи решаются при управлении виртуальной памятью.
      35. Что собой представляет идеальная стратегия решения задачи замещения.
      36. Какие применяются стратегии решения задачи замещения.
      37. Какие есть одноочередные дисциплины распределения ресурсов.
      38. Какие есть многочередные дисциплины распределения ресурсов.
      39. Какие есть модели организации ввода-вывода.

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)