ПРОЕКТ УЧЕБНОГО ПОСОБИЯ

ОСНОВЫ СИСТЕМНОГО ПРОГРАММИРОВАНИЯ

Семейство ОС UNIX, интерпретатор shell, язык C

Санкт-Петербург, 2018

Оглавление

Благодарности

Выражается особая благодарность людям, вкладу которых обязано настоящее пособие:

- Горской Александре Андреевне — за выборку, редактуру и верстку материалов первой версии пособия;
- Ховалкиной Ксении Николаевне — за работу с аудио материалами и набором машинописного текста;
- Кирееву Валерию Юрьевичу — за активное участие в записи лекционных материалов;
- Афанасьеву Дмитрию Борисовичу — за блестящее наставничество, внимание, чуткость и любовь, с которой он нас всех обучал;
- всем остальным лицам, мотивировавшим к написанию этого пособия.

Часть I

Системное программное обеспечение

Глава 1

Операционная система

1.1 Роль операционной системы

Операционная система предоставляет пользователю следующую функциональность:

Многозадачность

Позволяет одновременно выполнять несколько программ, изолируя их в разных адресных пространствах.

Виртуализация памяти

Операционная система абстрагирует программиста от физического адресного пространства, позволяет использовать больше адресного пространства, чем доступно оперативной памяти, а также изолирует программы, размещая их по непересекающимся адресам.

Управление устройствами

Система позволяет нам абстрагироваться от взаимодействия с устройствами,

беря эту задачу на себя.

В самом деле, было бы сложно каждый раз самим думать о том как, например, спозиционировать читающую головку на жестком диске или как вывести что-то на принтер.

Обработка прерываний

Операционная система реагирует на события, которые являются внешними по отношению к процессу (например, запрос на ввод-вывод). Такие события называются прерываниями.

Расширение набора операций, доступных программам

Операционная система расширяет набор команд, доступных программам. На уровне операционной системы появляются системные вызовы — обращения к функциям, которые реализованы в ядре операционной системы.

1.2 Типовая ОС UNIX

Архитектура операционной системы UNIX является многоуровневой [1]:

На самом внутреннем уровне располагается ядро. Функции ядра доступны через интерфейс системных вызовов. На среднем уровене находятся системные утилиты, демоны, драйверы, протоколы. На внешнем уровне — прикладные программы пользователя.

1.3 Ядро типовой ОС UNIX

Ядро — это центральная часть операционной системы. Ядро загружается и работает в младшей части адресного пространства оперативной памяти, таким образом, чтобы виртуальные и физические адреса соответствовали друг другу.

Ядро является своего рода большим диспетчером, обычной программой, которая работает на процессоре. Системные вызовы — механизм обращения к функциям ядра. В адресном пространстве ядра располагаются обработчики событий, сигналов, прерываний, исключений.

При переходе из режима задачи в режим ядра система разрешает доступ к этим адресам, а при обратном переходе — запрещает.

1.3.1 Подсистемы ядра

Ядро операционной системы UNIX состоит из следующих подсистем:

- **Файловая подсистема**

  Эта подсистема определяет унифицированный интерфейс доступа к файлам. Отвечает за такие процессы, как создание, удаление, чтение и запись файлов. Также файловая подсистема обеспечивает контроль прав доступа к файлам.

- **Подсистема управления процессами**

  Подсистема управления процессами занимается планированием процессорного времени, инициализацией и завершением процессов. Отвечает за синхронизацию и межпроцессное взаимодействие.

- **Подсистема ввода-вывода**

  Данная подсистема берет на себя взаимодействие с драйверами устройств и обеспечивает буферизацию данных.

1.3.2 Исключения, прерывания, системные вызовы

1.3.2.1 Исключения

Если при выполнении программы процесс пытается выполнить некорректную операцию, например, деление на ноль или доступ к участку памяти, который ему недоступен, то генерируется исключение. Исключение переключает процессор в режим ядра и вызывает обработчик исключений.

Обработчик исключений, в свою очередь, должен определить, может ли процесс восстановиться после ошибки. Если да, он выполняет действия по восстановлению и переводит процесс обратно в режим задачи, не сообщая о произошедшей ошибке.

Иначе выполняются действия по аварийному завершению работы процесса.

1.3.2.2 Прерывания

Прерывание —

> это приостановка выполнения программы по специальному сигналу.

Механизм прерываний нужен для оповещения системы об асинхронных событиях, таких, как нажатие клавиши на клавиатуре.

При выработке прерываний выполняется переход в режим ядра и поиск обработчика прерывания в IDT — Interrupt Descriptor Table. После исполнения обработчика прерываний выполняется обратный переход — в режим задачи.

Разница между исключениями и прерываниями состоит в том, что исключения вырабатываются процессором, а прерывания — периферией.

1.3.2.3 Системные вызовы.

Системный вызов —

> это запрос пользовательского процесса на выполнение какой-либо функции ядра от имени этого процесса.

Процессы не могут напрямую обращаться к ядру и должны использовать интерфейс системных вызовов. После того как процесс производит системный вызов, запускается специальная последовательность команд (называемая переключателем режимов), переводящая систему в режим ядра, а управление передается ядру, которое обрабатывает операцию от имени процесса.

В чем разница между прерываниями и системными вызовами?

При выполнении программного прерывания (int) у вас сразу же происходит переход к обработчику этого прерывания по вектору, который указан в инструкции. Код всех обработчиков лежит в оперативной памяти по заранее известным адресам, которые были выбраны разработчиками архитектуры. Внутри этих обработчиков и происходит обработка прерываний.

Надо понимать, что когда вы вызываете прерывание, происходит очень много аппаратной работы по сохранению контекста. Это довольно дорогая операция, поэтому со временем решили отказаться от использования “int

0x80” и использования механизма прерываний для системных вызовов в пользу новой инструкции syscall.

Инструкция syscall позволяет перейти к обработчику, который расположен в пользовательском пространстве, и попытаться обработать системный вызов, который вы хотите совершить. Только потом, если что-то понадобится от системы, можно обратиться к ней непосредственно.

Глава 2

Концепция системного программирования

Говоря о системном программном обеспечении, нужно понимать, что это составное понятие, которое включает в себя:

- языки СПО;
- системные вызовы;
- ввод-вывод;
- процессы и потоки.

Далее рассмотрим подробнее каждый из этих пунктов.

2.1 Языки системного программирования

Программирование можно глобально разделить на два вида: прикладное (для конечных пользователей) и системное (написание программ, которые

будут составлять систему саму по себе).

Языки системного программного опеспечения включают в себя языки, которые позволяют писать непосредственно программы для аппаратного уровня, — ассемблер и C.

Языку Си будет посвящена большая часть этого курса.

2.2 Системные вызовы

Системный вызов это некий механизм, который позволяет обратиться к системе. Механизм системных вызовов позволяет обращаться к функциям, которые реализованы в ядре операционной системы.

2.3 Ввод-вывод

2.4 Процессы и потоки

Глава 3

Работа в ОС UNIX

3.1 Терминал

Как до популяризации графических оболочек, так и после, базовой пользовательской средой является терминал.

В общем случае, терминал — это точка входа пользователя в систему, обладающая способностью передавать текстовую информацию. Операционная система обменивается с терминалом через последовательный интерфейс - терминальную линию.

В роли терминала может выступать как отдельное устройство, так и программа. Нам важно знать то, что при включении терминала запускается процесс авторизации пользователя, после которого запускается командный интерпретатор.

Терминал принимает текст и управляющие последовательности от устройств ввода (например, клавиатуры) и передает их командному интерпретатору.

3.1.1 Канонический и неканонический режимы ввода

Терминал умеет работать только в двух режимах — каноническом и неканоническом. Различаются они тем, что в неканоническом режиме введенный пользователем текст сразу же посимвольно отправляется на обработку операционной системе, а в каноническом режиме введенный пользователем текст сохраняется в так называемом line-буфере — буфере терминала — и отправляется в ОС только после нажатия клавиши enter.

Если проводить аналогию с написанием терминала для базовой ЭВМ, то в каноническом режиме был бы реализован буфер для хранения строки перед отправкой ее в БЭВМ. Раньше смысл этого заключался в том, чтобы сократить количество прерываний, потому что можно было вызвать одно прерывание “строка готова”, и процессор начинал эту строчку читать. С ускорением вычислительных ресурсов, увеличением их количества и введением таких вещей, как DMA (Direct Memory Access — прямой доступ к памяти), проблема себя исчерпала, и поэтому отпала необходимость в line-буфере.

Возвращаясь к каноническому и неканоническому режиму, помним, что мы либо вводим построчно, либо посимвольно. Разницу в этом мы прочувствуем несколько далее.

3.2 Командный интерпретатор

Командный интерпретатор или командная оболочка («shell») —

программа, являющаяся основным посредником между пользователем и операционной системой.

По сути, роль интерпретатора заключается в выполнении команды/набора команд, введенных пользователем (чаще всего — запуск других программ).

Важно: Командный интерпретатор, в отличие практически ото всех остальных программ, работает в неканоническом режиме и читает по одному символу.

3.3 Программа, утилита, команда

Приведем описания трех понятий — программы, утилиты и команды, которые часто вызывают путанницу.

Программа —

общее название, любой исполняемый код, реализующий какой-либо алгоритм.

Утилита —

программа очень узкой специализации (обычно ориентированная на выполнение одной или нескольких конкретных задач).

Команда —

указание командному интерпретатору, что нужно делать.

Часть II

Командный интерпретатор shell

Глава 4

Основные понятия

4.1 Командный интерпретатор

Как уже было сказано ранее, командный интерпретатор (“shell“) — программа, являющаяся основным посредником между пользователем и операционной системой.

Самыми частовстречаемыми оболочками являются Bourne shell (sh), Korn shell (ksh), Bourne Again shell (bash) и C shell (csh).

Важно: В этом курсе мы будем рассматривать все понятия, опираясь на работу самой первой из созданных оболочек - Bourne shell (sh).

4.1.1 Приглашения командной строки (prompt string)

Для того, чтобы помочь пользователю понять, чего ожидает от него интерпретатор, существуют так называемые prompt string (или приглашения командной строки). Все они имеют название по первым буквам - PS1, PS2,

PS3, PS4 и являются частью интерактивного режима ввода.

Их можно переопредилить в файлах инициализации (например, .profile).

$PS1 выводится в качестве для ввода команд. Туда можно помещать полезную информацию, такую, как текущий каталог, имя пользователя, под которым вы зашли, и так далее.

Очень часто в качестве последних символов в нем можно увидеть “$ <пробел>“ (для обычного пользователя), либо “# <пробел>“ (для пользователя, обладающего root-правами).

$PS2 выводится в случае, если мы начали вводить команду, но по какой-то причине не закончили ввод и нажали Enter.

Самое распространенное значение этой переменной - “>“.

$PS3 выводится тогда, когда оператор select ожидает ввода значений.

$PS4 выводится в начале каждой строки вывода, когда сценарий вызывается с ключом -x.

4.2 Принципы исполнения инструкций

4.2.1 IFS

Многие программы, с которыми вам предстоит работать, работают в каноническом режиме, и терминал по умолчанию работает в этом режиме. Еще раз, каждая строчка попадает в программу после нажатия Enter. Говоря о командном интерпретаторе, надо помнить, что он, в отличие практически от всех остальных, программ работает в неканоническом режиме и читает по одному символу.

После нажатия пользователем клавиши ввода и окончания всех подстановок командный интерпретатор производит синтаксический разбор введенной строки и разделяет ее на отдельные слова с помощью специальных разделителей, хранимых в переменной IFS.

Если вы не устанавливаете значение IFS, то оно по умолчанию содержит пробел, табуляцию и перенос на новую строку. Как уже сказано, в IFS может храниться несколько значений.

Например

```
ag@helios:/home/ag$ echo     123
123
```

Проигнорирует пробелы перед строкой 123.

4.2.2 Группировка команд и запуск команд в subshell

Фигурные скобки

Фигурные скобки используются для группировки команд. Перечисленные в **{ command1; command2; ... }** исполняются текущим командным интерпретатором.

Другая возможность использования — список возможных вариантов в **{ var1,var2, ... }**.

Например

```
ag@helios:/home/ag$ touch file_{a,b,c}
```

Создаст в текущем рабочем каталоге файлы с именами file_a, file_b, file_c.

Запуск команд в subshell

При перечислении списка команд в круглых скобках **$(<command>)** или обратных кавычках (гравис) **‘ <command> ‘**, они исполняются в дочернем shell (или по-другому subshell или подоболочка). То есть этот список команд будет исполнен новым экземпляром командного интерпретатора.

Отметим, что значения переменных, определенных в subshell, не передаются родительской оболочке и недоступны ей.

Например

```
ag@helios:/home/ag$ ls -l $(echo mydir)
```

Или эквивалентное ему

```
ag@helios:/home/ag$ ls -l ‘echo mydir‘
```

Выведут содержимое директории mydir — сначала выполнится команда echo в subshell, потом на место вызова subshell подставится результат работы — строка “mydir“.

4.2.3 Разделители команд

Для реализации работы сразу с несколькими командами был придумал механизм разделения команд между собой с помощью специальных операторов называемых разделителями команд.
Стоит сразу же отметить, что разделители команд можно комбинировать в любом варианте, в любом количестве.

Важно: Помните, что команды интерпретируются слева направо.

Рассмотрим следующие разделители команд:

“;” разделитель для последовательного выполнения команд.

Например

```
1    ag@helios:/home/ag$ mkdir somedir ; ls -l ; touch somefile
```

Последовательно выполнит команды, независимо от результата их выполнения.

“|” разделитель для создания конвейера - неименованного канала канала между двумя командами. То есть выходной поток команды до предстоящей “|” будет направлен на вход команде, стоящей после “|”.

Например

```
1    ag@helios:/home/ag$ ls -l | wc;
```

Подсчитает количество строк, слов, символов в выводе информации о содержимом домашнего каталога.

Момент, о котором часто забывают при работе с конвейером. Программы в нем запускаются и выполняются параллельно. Канал, это вещь без буфера, т.е. данные должны одновременно читаться и писаться. Это приводит к тому, что для каждой программы необходимо создавать свой процесс.

“&” оператор, предназначенный для запуска команд в фоновом режиме, но может также использоваться как разделитель.

Например

```
ag@helios:/home/ag$ find / -name somename & echo "Hello"
```

Запустит команду find / -name somename в отдельнов фоновом процессе, а команда echo “Hello” будет запущена в обычном режиме и использует терминал (Выведет “Hello”).

Очень часто команды необходимо объединять разумно, то есть сложнее, чем способами описанными выше. Например, когда нам нужно, чтобы одна команда выполнилась только в случае успешного завершения другой команды. Это возможно с помощью операторов логической конъюнкции “И” и дизъюнкции “ИЛИ”.

Немного о том, как shell понимает, завершилась ли команда успешно. Для этого существует такое понятие как **код возврата** — число характеризующие успешность выполнения команды.

Код возврата успеха в shell, это всегда - 0. Если же команда завершилась неуспешно, мы можем посмотреть ее код возврата в переменной $?, чтобы узнать более точную причину ошибки, а не просто факт ее возникновения.

“&&” разделитель “И”. Команда стоящая после “&&” выполнится только в случае успешного выполнения команды, стоящей до “&&”.

Например

```
ag@helios:/home/ag$ rm file1 && echo “”Hello
```

“Hello“ будет выведено только в случае, если file1 был успешно удален.

“||” разделитель “ИЛИ”. Команда стоящая после “||” выполнится только в случае неуспешного выполнения команды, стоящей до “||”.

Например

```
ag@helios:/home/ag$ rm file1 || rm file2 && echo "Hello"
```

“Hello“ будет выведено только в случае, если был успешно удален хотя бы один из файлов file1, file2.

4.2.4 Glob-джокеры

В shell есть три glob-джокера. Это **“*”**, **“?”**, **“[]”**. Будьте осторожны и не путайте их с квантификаторами регулярных выражений.

Glob-джокеры — это специальные символы-шаблоны поиска, которые в некоторых случаях заменяются shell, в зависимости от их назначения.

***** означает любое количество любых символов

? — означает один любой символ

[] — работает как символьный класс (вхождение одного из символов, перечисленного в скобках). Если наоборот нужно исключить какие-либо символы, после [можно поставить символ ^

Например

```
ag@helios:/home/ag$ ls mydir/f?l*.sh
```

Выведет все файлы, содержащиеся в директории mydir, которые содержат последовательность “f“, один произвольный символ, символ “l“, любое количество любых символов, последовательность символов “.sh“.

4.2.5 Переменные

Внутри имен переменных shell могут использоваться буквы заглавного и прописного регистра, нижнее подчеркивание и любые цифры. Однако, первую позицию цифра занимать не может — переменные, начинающиеся с цифры, являются специальными переменными для позиционных параметров. О них мы поговорим в другом разделе.

Для присваивания переменной значения используется оператор присваивания “=>“. В качестве значения shell-переменной устанавливается строка символов. Чтобы получить значение переменной, используется символ “$“.

Например

```
ag@helios:/home/ag$ myvar=word
ag@helios:/home/ag$ echo $myvar
word
```

Присвоит переменной myvar значение word и выведет его на экран.

4.2.5.1 Обрезка переменных

Иногда есть необходимость обрезать часть значения переменной при её использовании. Для этого существуют символы подстроки # и %. Символ подстроки # позволяет обрезать содержимое переменной слева, а символ % - справа.

Запомнить можно так: # расположена на клавиатуре слева, % — справа.

Рассмотрим, как это работает на примерах.

Символ подстроки #

```
ag@helios:/home/ag$ myvar="airport is good"
ag@helios:/home/ag$ echo ${myvar#air}
port is good
```

Перед выводом значения переменной myvar shell увидит символ подстроки # и исключит из вывода первые три символа — “air“, если исходное значение переменной начинается с этих символов.

Символ подстроки %

```
ag@helios:/home/ag$ myvar="airport"
ag@helios:/home/ag$ echo ${myvar%port}
air
```

Перед выводом значения переменной myvar shell увидит символ подстроки % и исключит из вывода последние четыре символа — “port“, если исходное значение переменной заканчивается этими символами.

Не забывайте также о возможности использования glob-джокеров

```
ag@helios:/home/ag$ myvar="airport is good"
```

```
ag@helios:/home/ag$ echo ${myvar%port*}
air
```

Важно: Символы % и #, можно сказать, дадют только временный эффект. Значение самой переменной они не изменяют. Это можно сделать только использовав операцию присвоения.

Например

```
ag@helios:/home/ag$ myvar="airport is good"
ag@helios:/home/ag$ myvar=${myvar%port*}
```

4.2.6 Как shell интерпретирует команду

Коротко о том, как shell интерпретирует команду. После набора инструкции и нажатия клавиши ввода можно выделить следующие шаги:

1. подстановка:
   (a) результатов команд, выполненных в подоболочке;
   (b) позиционных параметров и переменных;
   (c) вычисление арифметических выражений;
   (d) путей (glob-джокеров);
   (e) разбиение на слова (в соответствии с IFS).
2. подстановка алиасов;
3. поиск команды сначала по встроенным в shell, а в случае ненахождения — по путям, указанным в переменной $PATH;
4. выполнение команд согласно расположению и свойствам разделителей

команд.

Например

```
ag@helios:/home/ag$ ls -l $(echo mydir) && rmdir mydir
```

Глава 5

Скрипты shell

Программы, написанные на языке командного интерпретатора, называют сценариями или скриптами.

Скрипт может быть передан командному интерпретатору с помощью специального командного файла. Особенностью этого специального файла будет являться наличие в первой строке конструкции вида: **#!/bin/sh** (#!<путь к командному интерпретатору, из которого мы хотим запустить скрипт).

Первые два символа называются shebang. Есть две возможных трактовки этого названия. Первая — от слова Shell, второй вариант ближе к символьной трактовке: Sharp(#) и bang(!).

5.1 Именование и запуск скриптов

В случае shell, расширение — атавизм из Windows. То есть для shell .sh, .bash, .ksh — не более, чем окончание имени файла. Тем не менее, добавлять к имени файла .sh (или название другого shell), можно для того, чтобы

пользователь понимал, что это shell-скрипт. Например, если планируется передавать этот скрипт широкому кругу пользователей.

Сохранив скрипт, можно запустить его следующим образом:

```
ag@helios:/home/ag$ chmod 755 myscript
ag@helios:/home/ag$ ./myscript 1 2 3
```

Важно: Перед первым запуском не забудьте дать себе права на выполнение этого скрипта с помощью chmod.

5.2 Позиционные параметры

Позиционные параметры —

> это аргументы, передаваемые скрипту при запуске из терминала, и имя самого скрипта.

Например

```
ag@helios:/home/ag$ ./myscript 16 name 4
```

myscript - имя скрипта, 16, name, 4 - аргументы, переданные скрипту.

Внутри самого скрипта можно посмотреть с помощью позиционных переменных: **$0, $1, $2, $3,...**, где $0 – имя файла запущенного скрипта
$1, $2, $3.. — аргументы, передаваемые скрипту.

Важно: аргументы, следующие за $9, должны заключаться в фигурные скобки, например: ${10}, ${11}, ${12}. Отсутствие фигурных скобок может привести к интерпретированию только первой цифры в качестве позиционного параметра (Например $89 как $8 и цифра 9).

5.3 Специальные переменные

Говоря о позиционных параментрах, нельзя забывать две переменных окружения — “@” и “*”. Это такие переменные, которые позволяют обратиться ко всем позиционным параметрам, начиная с первого.

При написании без двойных кавычек (echo $* и echo $@) эти переменные выдают одинаковый результат - подставляют вместо себя позиционные параметры разделенные пробелом. Разница станет значительной лишь при использовании двойных кавычек:

echo “$*” выведет одну большую строку в кавычках, разделенную первым символом из IFS.

echo “$@” выдаст набор разделенных пробелом позиционных параметров (по сути массив,с которым мы сможем работать). Этот момент всегда нужно помнить.

Рассмотрим пример с установкой значения IF в :

Файл myscript

```
#!/bin/sh
IFS=:
echo "$*"
echo "$@"
```

shell

```
  ag@helios:/home/ag$ ./myscript test 67 value
  test:67:value
  test 67 value
```

На самом деле, специальных переменных довольно много, рассмотрим некоторые из них:

$- содержит ключи, передаваемые shell (фактически опции задаваемые с помощью set).

$_ содержит последний аргумент предыдущей выполненной команды.

$# содержит число позиционных параметров.

$? содержит код возврата предыдущей команды.

$! содержит идентификатор процесса, который был последним свернут в бэкграунд группу. Если вы запустили два процесса, то в переменной $! будет PID последнего процесса, который был запущен.

Глава 6

Встроенные возможности и конструкции shell

Как мы помним shell не язык программирования, а автоматизатор для запуска команд. В этой главе мы рассмотрим, что всё же приближает его к языкам программирования.

6.1 Математические операции

Командный интерпретатор поддерживает набр следующих математических операций:

Обозначение	Наименование
+	Сложение
-	Вычитание
*	Умножение
/	Деление
%	Остаток от деления
<<	Сдвиг влево
>>	Сдвиг вправо
&	Логическое И
|	Логическое ИЛИ
^	Исключающее ИЛИ

6.2 Условные кострукции, циклы, функции, комментарии

6.2.1 Условные кострукции

Для ветвления программ в shell существуют условные конструкции IF и CASE.

IF...THEN...ELSE

```
IF ( <condition> )
  THEN <actions>
```

```
ELIF ( <alternative condition> )
  THEN <actions>
ELSE
  <actions>
FI
```

В качестве условия выполнения — <condition> — принимает набор команд (Например, через &&). Действие после THEN будет выполнено только в случае, если код возврата последней команты — успех.

Важно: Помним, что при написании IF и THEN в одной строке они должны разделяться “;”, так как являются разными командами

То есть:

```
IF ( <condition>) ; THEN <Actions>
ELIF ( <alternative condition> ) ; THEN <actions>
ELSE <actions>
FI
```

CASE

```
CASE (<string>) IN
    pattern -1)
   <actions>
            ;;
    pattern -2)
     <actions>
            ;;
    pattern | pattern)
     <actions>
            ;;
ESAC
```

Принимает строку и в случае соответствие ее одному из установленных шаблонов совершает необходимые действия. По сути, аналог switch.

Важно: Не забываем о возможностях использования glob-джокерах в CASE. Так * будет обозначать не вошедшие в шаблоны случаи.

Заметка: завершающие слова в IF и CASE - просто написание их же в обратном порядке (IF - FI, CASE - ESAC)

6.2.2 Циклы

Довольно часто условные конструкции можно опускать за счет использования “&&” или “||”, а вот такие вещи как циклы имитировать просто так не получится

Всего в shell три вида циклов - while, for и until, рассмотрим их подробнее.

WHILE

```
WHILE ( <condition> )
  DO
    <actions>
  DONE
```

Код внутри do - done будет выполняться пока условие истинно

UNTIL

```
UNTIL ( <condition> )
  DO
    <actions>
  DONE
```

Код внутри do - done будет выполняться пока условие ложно

FOR

```
FOR var IN ( <list of values> )
  DO
    <actions>
  DONE
```

Обеспечивает выполнения столько раз, сколько значений в списке значений,

при этом переменная var последовательно принимает значения из списка значений при каждой новой итерации

Такая форма for в принципе не вызывает никаких вопросом. Но что будет, если опустить список значений? Дело в том, что это позволит неявно пройти по позиционным параметрам переданным shell, то есть var будет принимать уже их значения

BREAK и CONTINUE

Вспоминая си, у нас есть возможность перейти к следующей итерации цикла с помощью continue и закончить, с помощью break. В shell есть похожие слова.

Отличие BREAK в shell в том, что он позволяет выходить не только из последнего цикла, в котором он написан, но и из любого уровня вложенности циклов.

BREAK N, где N — количество уровней циклов, из которых мы выйдем.

Ключевое слово CONTINUE позволяет переходить заново к проверке условия (в начало цикла), то есть работает также как, например, в языке Си.

6.2.3 Функции

Когда мы пишем shell скрипты мы можем воспользоваться такой вещью как функцию, чтобы ее задать надо использовать следующий синтаксис:

```
function func {
  echo "Inspiration unlocks the future."
}
```

Ключевое слово function не является обязательным, его можно опускать.

Перечислять аргументы в прототипе не нужно потому как все аргументы, которые вы передадите этой функции будут доступны как позиционные параметры.

Пример функции и ее вызова с параметрами

```
function func {
  echo $1 $2
}

func $myvar1 $myvar2
```

6.2.4 Комментарии

Когда вы пишите shell скрипты лучше бы оставлять комментарии, особенно, если вы собираетесь с ним работать через год. Это хороший стиль.

Комментарии в shell начинаются с решетки (#) и бывают только однострочными.

Например

```
echo "Hello, my friend" # this is comment
```

Строка после символа # будет проигнорированна shell.

Глава 7

Потоки ввода-вывода

Внутри каждой программы у ОС UNIX есть понятие потоков ввода-вывода. Фактически потоки - это файлы, которые связаны с программой с помощью специального числа, называемого файловым дескриптором. Потоки предназначены для ввода и вывода данных.

7.1 Стандартные потоки ввода-вывода

В классических системах юникс дескрипторов всего 256, причем первые три из них зарезервированы и по умолчанию ассоциированы с терминалом (то есть каждая запущенная из shell программа будет ждать ввода с клавиатуры терминала или выводить результат/ошибки на экран терминала).

Номера и назначения этих потоков представлены здесь:

Номер	Название	Назначение
0	stdin	Стандартный поток ввода
1	stdout	Стандартный поток выводы
2	stderr	Стандартный поток ошибок

7.2 Перенаправление потоков

Для удобства и расширения возможностей работы с потоками в юникс существует механизм из перенаправления. Рассмотрим его синтаксис:

i > filename перенаправление потока с дескриптором i в файл filename с перезаписью содержимого filename

i >> filename перенаправление потока с дескриптором i в файл filename с дозаписью содержимого filename

i >| filename также перенаправление потока с дескриптором i в файл filename. Однако, в случае, если файл существует, он не будет перезаписан

< filename получение стандартного потока ввода из filename

Например

```
ag@helios: /home/ag$ wc -l < myfile
```

Подсчитает количество строк в myfile

<< “string” shell будет запрашивать ввод у пользователя, пока пользователь не передаст на вход строку, указанную после символов <<

Например

```
ag@helios: /home/ag$ cat -n << end
> I want to see in cat -n output
> only lines, which I've sent
> before line, contained only word
> end
    1  I want to see in cat -n output
    2  only lines, which I've sent
    3  before line, contained only word
```

Выведет нумерованные строки, введенные до строки, содержащей только слово “end“

i <> filename открывает файл filename на чтение и запись и связывает его с дескриптором i

i >& j поток с дескриптором i будет указывать туда же куда и поток с дескриптором j, в Си — системные вызовы dup (2) и dup2 (2)

Для перенаправления потоков также активно используется механизм конвееров, описанный в разделе “Разделители команд”

Глава 8

set и env

8.1 Команда set

set —

встроенная в shell команда, она предназначена для установки или сброса ключей и позиционных параметров.

Например

```
1  ag@helios:/home/ag$ set word1 word2 word3
```

Установит значения 1-го, 2-го и 3-го позиционных параметров, соответственно.

Процесс экспортирования необратим. Единственный способ сделать эту переменную не экспортированной это удалить эту переменную т..е сделать операцию unset.

Приведем некоторые режимы set:

- **set -v** – выводит на терминал текст исходной команды, перед ее выполнением;
- **set -x** – выводит на терминал текст интерпретированной команды, перед ее выполнением;
- **set -vi/emacs** – устанавливает в shell режим управления как в соответствующих редакторах;
- **set -o ignore EOF** – игнорировать конец файла;
- **set -o no exec** – запретить выполнение команд в текущем shell;
- **set -o no clobber** – аналог >|.

Важно: Символ - перед опцией означет включение режима, символ + – выключение

8.2 Утилита env

env —

отдельный файл-утилита, находится в user/bin. Происходит от слова environment - окружение. Утилита env позволяет запускать команды не с текущим окружением, а с любым созданным вами.

У env есть всего один ключ -i (от слова initial - начальный). Этот ключ значит, что все окружение запускаемой команды будет перечислено в аргументах env. Других переменных в окружении у этой команды не будет.

Тонкий момент с утилитой env и ее использовании. Очень часто ее используют в shebang. Например, если вы не знаете где в системе пользователя лежит какой-то язык программирования и знаете, что есть каталог /usr/bin/env,

то вы можете написать в shebang следующую конструкцию: #!/usr/bin/env python. Это позволит нам запустить python не зная где он расположен на системе. Фактически написать скрипт, который будет легко переносим.

Часть III

Язык Си в системном программировании

Целью этой части не является изучение языка Си, она носит только вспомогательныхй характер. В основном предназначена для освежения и восполнения пробелов в знаниях о языке Си, иззученном в рамках других курсов.

В этой части также будут рассмотрены некоторые особенности системного программирования на языке Си. В том числе в операционной системе UNIX.

Глава 9

Программы на языке Си

В этой главе мы рассмотрим некоторые особенности системного программирования на языке Си, структуру С-программы и способы уточнения причин ошибок, возникающих в ходе ее работы.

В качестве основного примера возьмем следующую программу:

Листинг main.c

```
#include <stdlib.h>
#include <fcntl.h>
#include <errno.h>

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

  if (open(argv[1], O_RDONLY) < 0) {
    perror("main");
    return EXIT_FAILURE;
  }

  close(argv[1]);

  return EXIT_SUCCESS;
}
```

Ее назначение — определить существует ли доступный для чтения файл с именем, поданным первым аргументом на вход программе.

9.1 Заголовочные файлы

В первых строках нашей программы указаны три директивы:

main.c

```
1       #include <stdlib.h>
2       #include <fcntl.h>
3       #include <errno.h>
```

В нашем случаее они нужны для использования open (2), close (2) и perror (3), а также макросов EXIT_FAILURE И EXIT_SUCCESS.

Файлы с расширением .h, указанные внутри <> у директивы #include, являюся заголовочными (или подключаемыми) и могут содержать:

- прототипы функций;
- структуры;
- союзы (union);
- перечисления (enum);
- макросы;
- объявления типов (typedef);
- глобальные переменные.

Сформулируем общее определение для заголовочных файлов:

Заголовочный файл —

файл, содержимое которого при компилляции автоматически добавляется препроцессором в исходный текст программы на место специальной директивы.

Важно: Заголовочные файлы **не должны** описания функций и системных вызовов (только их прототипы).

9.1.1 Часто используемые заголовочные файлы UNIX

Приведем заголовочные файлы, наиболее часто используемые в системном программировании в опреационной системе UNIX.

Имя	**Содержимое**
unistd.h	объявления UNIX
stdio.h	стандартный ввод/вывод
fcntl.h	операции с файлами (например: open)
sys/types.h	системные типы
sys/stat.h	системные статусы
errno.h	errno и и директивы с определением ошибок

Более подробно узнать об их назначении можно узнать использовав команду **man**.

Например

```
ag@helios:/home/ag$ man stdlib
ag@helios:/home/ag$ man stdlib.h
```

9.2 main-функция

Исполнение любой программы, написанной на языке Си, начинается с функции main. Эта функция — точка входа в программу.

Классическая библиотека libc подразумевает такой прототип функции main и примерно такое тело:

main.c

```
int main(int argc, char *argv[], char *envp[]) {

  /* Actions */

  return EXIT_SUCCESS;
}
```

9.2.1 Аргументы main-функции

Как вы уже заметили, в классическом прототипе функции main содержатся три аргумента: argc, *argv[] и *env[]. Рассмотрим каждый из них подробнее.

- **int argc** arguments count

  Содержит количество переданных программе аргументов, включая имя программы.

- **char *argv[]** argument vector

  Является указателем на массив, в котором содержится каждый из параметров переданных программе аргументов, включая имя программы (агрумент 0).

- **char *envp[]** environment parameters

Является указателем на массив переменных окружения, в котором запущена программа.

9.3 Код возврата

В исходной программе мы написали два return: EXIT_SUCCESS и EXIT_FAILURE. EXIT_SUCCESS и EXIT_FAILURE — это макросы, определенные в заголовочном файле stdlib.h (обычно эквивалентные 0 и 1 соответственно).

main.c

```
int main(int argc, char *argv[], char *envp[]) {
  if (open(argv[1], O_RDONLY) < 0) {
    perror("main");
    return EXIT_FAILURE;
  }

  close(argv[1]);

  return EXIT_SUCCESS;
}
```

Дело в том, что после отработки main-функции, return возвращает свое значение чему-то, что инициировало вызов этой самой main. Таким образом, значение return () позволяет инициатору вызова определить корректность (или некорректность) завершения функции. Значение, определяемое в return, принято называть кодом возврата.

Код возврата —

> число, возвращаемое родительскому процессу при завершении дочернего, позволяющее сделать выводы о результате выполнения программы.

Иными словами, код возврата, позволяет кому-то вызывающему программу определить, завершилась ли она корректно, а если нет, то принять какие-нибудь меры (например, изменить входные данные для программы).

Важно: В отличие от shell, в Си код возврата в случае успеха имеет значение 0 или положительного числа, а в случае ошибки отрицательное значение.

Как мы уже заметили, в приведенной в начале главы программе в return вместо чисел используются макросы EXIT_SUCCESS и EXIT_FAILURE из заголовочного файла stdlib.h. Как же правильнее?

Указывать в return число

```
1  int main(int argc, char *argv[], char *envp[]) {
2      /* Actions */
3      return 0;
4  }
```

или

Указывать в return идетификатор

```
1  int main(int argc, char *argv[], char *envp[]) {
2      /* Actions */
3      return EXIT_SUCCES;
4  }
```

На самом деле, на подавляющем большинстве систем макрос EXIT_SUCCESS будет также равен нулю. Однако, мы должны помнить, что если мы будем запускать систему на каком-нибудь контроллере, то его успешный код возврата может быть отличным от нуля.

Таким образом, использование данного макроса делает код более партируе-

мым и кроссплатформенным.

9.4 Ошибки и стандартизация ошибок

Теперь поговорим об ошибках, возникающих при работе с системными вызовами, и о том, как можно узнать, почему тот или иной вызов не завершился успешно.

Системный вызов является либо вызовом инструкции int, либо вызовом инструкции syscall. При выполнение этой инструкции, система возвращает статус системного вызова через регистр AX вызова (то есть код возврата). Помним, что неотрицательный код возврата, это успешное выполнение, отрицательный (чаще всего -1) — ошибка.

Однако, нам часто хочется узнать более точную причину произошедшей ошибки, а не просто факт ее возникновения. Для осуществления этой возможности и была придумана специальная **переменная errno**. Эта переменнтая содержится в заголовочном файле errno.h и имеет тип external int.

Помимо errno в <errno.h> хранятся также и директивы с определением ошибок и их кодов.

Пример содержимого errno.h

```
...
#define ENODEV      19  /* No such device */
#define ENOTDIR     20  /* Not a directory */
#define EISDIR      21  /* Is a directory */
...
```

Как это работает?

Дело в том, что в случае возникнавения ошибки, системный вызов не только возвращает отрицательное значение. Он также устанавливает значение errno в зависимости от того, по какой причине эта самая ошибка случилась (если ошибки не произошло, значение errno не изменяется).

В чем еще преимущество errno?

В разных ОС коды возврата системных вызовов они чуть-чуть отличаются друг от друга, поэтому если наша программа анализировала содержимое AX регистра, то мы бы устали для каждой ОС делать исключения. Libc предлагает нам унифицировать не только ввод-вывод, но и взаимодействие системных вызовов.

Как это работает? Чтобы программистам на языке Си было проще, код возврата функции – обертки системного вызова приводится к форме отрицательного числа уже средствами самой libC.

9.4.1 perror (3) и strerror (3)

В реальном коде, для быстрого получения описания ошибки используются функции prerror и strerror. Они работают, анализируя текущее значение errno.

perror (3)

```
1  void perror(
2    const char *str
3  );
```

Выведет в STDOUT сообщение вида “str: <описание текущего значения errno>”

strerror (3)

```
char * strerror(
  int errnum
);
```

Вернет указатель на строку с описанием текущего значения errno

Например в нашей исходной функции, perror (3) используется, чтобы диагностировать ошибку, возникающую в функции main, если ОС не удалось открыть файл.

main.c

```
if (open(argv[1], O_RDONLY) < 0) {
  perror("main");
  return EXIT_FAILURE;
}
```

Глава 10

Компиляция программы

Как вы можете помнить из других курсов, компиляция программы нужна для получения исполняемого файла. Сам процесс компиляции состоит из этапов препроцессинга, трансляции и компоновки.

На этапе препроцессинга осуществляется подстановка исходного тела макросов на место их имен. Далее происходит трансляция полученного кода в язык более низкого уровня. Последний этап – компоновка, устанавливает связи между отдельными различными файлами и генерирует из них один - исполняемый.

10.1 Компиляторы

Представим некоорые компиляторы подходящие для языка Си:

- **cc** C compiler

- **gcc** GNU project C and C++ compiler

- **clang** C and Objective-C compiler

10.2 Makefile и утилита make

Для автоматизации сборки программ существуют две полезные вещи — утилита make и Makefile.

Утилита make предназначенна для автоматизации преобразования файлов из одной формы в другую. Правила преобразования задаются в скрипте с именем Makefile, который должен находиться в корне рабочей директории проекта).

Рассмотрим структуру Makefile на примере.

Makefile

```
CC=gcc
CFLAGS=-m64

all: $(PROJS)
  @echo Done!
main:
  $(CC) $(CFLAGS) -o $@ $(@:=.c)
```

В начале объявляются переменные, затем конструкция вида:

<цель> : <зависимости>
<список команд>

Утилита make

```
ag@helios:/home/ag $ make
gcc -m64 -o main main.c
Done!
ag@helios:/home/ag $ ./main
Inspiration unlocks the future
```

При команде **make <цель>** будут выполнены команды соответствующие метке <цель>, а также последовательно все зависимости(другие метки), если они не являются файлами.

В нашем примере make по умоляанию перейдет к цели all, увидит зависимость main и перейдет на эту метку. Далее так как зависимостей у main нет, make выполнит команду “gcc -m64 -o main main.c“, а затем вернется к метке all и выполнит команду echo.

Makefile имеет специальные собственные переменные.

Приведем некоторые из них:

- $@ — имя текущей цели;
- $< — имя первой зависимости Makefile;
- $^— имена всех зависимостей.

Часть IV

Файлы и файловая подсистема

Глава 11

Все есть файл! (Кроме потоков и ядра)

В современном мире существует огромное количество самых различных устройств, предназначенных для ввода и вывода информации. Было бы тяжело, если бы с каждым из этих устройств пришлось работать по-разному. Для решения этой проблемы создателями UNIX была выбрана концепция “Everything is a file“ или “Всё есть файл“.

В этой концепции, всё, с чем бы не взаимодействовала пользовательская программа, представляется для этой программе в виде файла. Это даёт следующие возможности: мы просто пользуемся принтером, жестким диском или другим устройством как обычным файлом — пишем в него данные с помощью одного и того же файлового интерфейса, одних и тех же системных вызовов. В свою очередь, операционная система направляет данные по назначению.

Долгие споры приводят к тому, что четкого определения слову файл нет. В нашем курсе будем полагать, что:

Файл —

совокупность неких данных и метаинформации (метаданных), которая описывает эти данные.

Не будем претендовать, что оно всегда и везде будет соответствовать действительности, потому что на курсе операционных систем вам скажут, что файлом может называться, например, магнитный кусок ленты, на которой вы записали данные. И будут, безусловно, правы. Поэтому для себя запомним, что файл это некая такая абстрактная неведомая вещь, с которой мы можем условно каким-то образом взаимодействовать и по сути она нужна для работы с данными.

Важно: В заголовке есть некоторое уточнение - что всё есть файл, кроме потоков и ядра. Дело в том, что в юниксе еще есть много других вещей в том числе потоки (имеются в виду не streams, а threads) и ядро. Ядро само по себе, это работающая программа, которая позволяет пользовательским программам выполняться и работать с файлами. А потоки позволяют коду выполняться параллельно. Мы будем отталкиваться от того, что файл, все же, это некие данные и метаданные, которые эти данные описывают (надо понимать, что этих данных может как таковых не быть).

11.1 inode

Каждый файл в ОС UNIX описывается специальной структурой — индексным дескриптором (inode).

Индексный дескриптор —

структура, описывающая файл, содержащая метаданные о файле (служебную информацию, необходимую для обработки данных).

Приведем некоторы важные поля, содержащиеся в индексном дескрипторе:

- **mode** —

  тип файла и права на него;

- **nlink** —

  количество жестких ссылок;

- **uid** —

  идентификатор пользователя – владельца файла;

- **gid** —

  идентификатор группы – владельца файла;

- **size** —

  размер файла в байтах;

- **atime** —

  Время последнего доступа;

- **mtime** —

  время последнего изменения содержимого;

- **ctime** —

  время последнего изменения метаданных;

- **gen** —

  генерируемый номер новой inode (инкриментируется при каждом запросе идентификатора для нового файла);

- **addr[13]** —

  адреса содержимого файла.

Важно: Индексный дескриптор не содержит имени файла и его содержимого.

На самом деле в ОС UNIX существуют три типа inode —

- inode — хранится в оперативной памяти
- dinode — хранится на диске
- vnode — представлена в VFS (Виртуальной файловой системе). VFS нужна для обеспечения единообразного доступа к различным типам файловых систем.

11.2 Структура stat

Довольно часто нам требуется узнать информацию о каком-то файле. Какого типа информацию? Например, что это за файл, его права доступа. Фактически, ту информацию, которая хранится в inode. И эту информацию мы можем получить с помощью системного вызова stat (2). Называется он от "get status file"(получить статус файла).

stat —

структура для хранения метаданных файла, полученных системным вызовом stat (Сами метаданные хранятся в структуре индексного дескриптора).

Некоторые поля структуры stat

Поле	**Назначение**
mode_t st_mode	Тип файла и права на него
ino_t st_ino	Номер индексного дескриптора
dev_t st_dev	Номер устройства, на котором хранится файл
NLink_t st_nlink	Количество жестких ссылок
uid_t st_uid	UID
gid_t st_gid	GID
off_t st_size	Размер файла в байтах
time_t st_atime	Время последнего доступа
time_t st_mtime	Время последней модификации содержимого
time_t st_ctime	Время последней модификации метаданных
long st_blksize	Оптимальный размер для ввода-вывода
long st_blocks	Число размещенных блоков хранения

Для получения stat существует следующее семейство системных вызовов:

stat (2)

```c
#include <sys/types.h>
#include <sys/stat.h>

int stat(
  const char *path,
  struct stat *st
);
```

Через имя файла (для обычного файла).

lstat (2)

```c
#include <sys/types.h>
#include <sys/stat.h>

int lstat(
  const char *path,
  struct stat *st
);
```

Через имя файла (для символьной ссылки).

fstat (2)

```c
#include <sys/types.h>
#include <sys/stat.h>

int fstat(
  int filedes,
  struct stat *st
);
```

Через файловый дескриптор

Эти системные вызовы возвращают ноль, в случае успеха, либо код ошибки.

11.3 Файловый дескриптор

Немного о том, как программы отличают файлы, с которыми они взаимодействуют. Для обращения программы к файлу в ОС UNIX существует специальная структура — файловый дескриптор.

Файловый дескриптор —

> структура описывающая файл, с которым взаимодействует программа.

Каждый раз, когда программа (процесс) создает новый поток ввода-вывода (например, при открытии файла), ядро создает новый файловый дескриптор в таблице открытых файловых дескрипторов. Таблица файловых дескрипторов существует для каждого отдельного процесса и находится в его u_block области, то есть один и тот же дескриптор у разных процессов может ссылаться на разные файлы.

Номер дескриптора	Имя файла	Флаги доступа
...	...	...
42	/etc/passwd	O_RDWR
43	/home/ag/secrets	O_RDONLY

Таблица 11.1. Структура таблицы открытых файловых дескрипторов

Как можно видеть, каждая запись о файловом дескрипторе в таблице файловых дескрипторов имеет:

- номер — целое неотрицательное число не более константы OPEN_MAX,

определяемой ОС;

- символьное имя (путь) файла в файловой системе;
- флаги доступа к файлу (каким образом данный процесс может с этим файлом взаимодействовать).

Важно: Также в таблице файловых дескрипторов процесса по умолчанию уже открыты стандартные потоки ввода, вывода и ошибок. Они имеют номера файловых дескрипторов 0, 1 и 2 соответственно (в стандарте POSIX.1 используются константы STDIN_FILENO, STDOUT_FILENO и STDERR_FILENO).

Вы можете точно также закрыть, открыть только на чтение и т.д. любой из стандартных потоков.

После создания файлового дескриптора ОС возвращает процессу номер этого дескриптора в таблице открытых файловых дескрипторов для данного процесса. В дальнейшем, процесс может обращаться к файлу по номеру возвращенного файлового дескриптора, тем самым абстрагируясь от файлов, с которыми он взаимодействует.

Иногда для упрощения речи говорят: открой 8-ой дескриптор для записи файла. Это не совсем верно. Правильнее было сказать открой дескриптор с номером 8 на запись в файл, так как файловый дескриптор это все-таки структура, а не число.

11.3.1 Работа с файловым дескриптором

Для создания дубликата (копирования) существующего файлового дескриптора используются ситемные вызовы dup(2) и dup2(2) (от слова duplicate). Оба этих вызова в качестве первого аргумента принимают номер уже открытого файлового дескриптора. Разница между этими двумя вызовами

лишь в том, что dup2 позволяет задать значение желаемого номера нового файлового дескриптора во втором агрументе.

dup(2)

```
int dup(
  int fildes /* opened file descritor number */
);
```

Возвращает: дубликат файлового дескриптора или код ошибки.

dup2(2)

```
int dup(
  int fildes /* opened file descritor number */,
  int fildes2
);
```

Возвращает: дубликат файлового дескриптора или код ошибки.

В каких случаях может произойти ошибка?

Очевидно, в случае если файловый дескриптор указывает несуществующий, не открытый поток. Если функция dup(2) не находит свободного места в таблице дескрипторов открытых файлов, то тоже вернется ошибка, что нет свободного места.

11.3.2 fcntl

Теперь мы поговорим о том, что не особо соответствует философии UNIX-way. UNIX-way — это философия юникс, одно из положений которой гласит: делай вещь, которая работает просто и решай какую-то одну задачу. В противовес этому есть такие решения, как **fcntl f (file) cntl (control)** — системный вызов, предназначенный для выполнения операций над файлом, используя его файловый дескриптор. Операция определяется аргументом

cmd.

Собственно функция fcntl позволяет работать с файловыми дескрипторами. Первый аргумент и есть файловый дескриптор, с которым мы работаем. Дальше, выполнять некие команды с этим дескриптором, которые мы передаем вторым аргументом, а дальше переменное количество аргументов, которые будут параметризовать запрошенную команду. Возвращает соответствующее значение той команды, которую мы запросили.

fcntl(2)

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

int fcntl(
  int fildes,
  int cmd, /* args */ ...
);
```

Управление флагами дескриптора файла

F_GETFD получает значения флагов дескриптора, описанных в <fcntl.h>, для указанного файлового дескриптора.

F_SETFD устанавливает значения флагов дескриптора для указанного файлового дескриптора, в соответствии с битами arg. Если флаг FD_CLOEXEC равен 0, то файловый дескриптор останется открытым после вызова exec, иначе - будет закрыт.

Управление флагами состояния файла

F_GETFL получает флаги статуса файла и режим доступа, для указанного файлового дескриптора.

F_SETFL устанавливает флаги статуса файла, согласно значению arg. Оставшиеся биты (режим доступа, флаги создания) в значении arg игнорируются.

Управление сокетами

F_GETOWN возвращает pid или pgid процесса, который получает сигнал SIGURG (появление срочных (urgent)) данных. Положительные значения относятся к pid, отрицательные, кроме -1 - к pgid.

F_SETOWN изменить pid или pgid процесса, ответственного за получение SIGURG. Положительное значение arg относится к pid, а отрицательные, кроме -1 - к pgid.

11.4 Типы файлов

В классическом юниксе всего семь типов файлов, каждый из них имеет свои особенности и назначение. Кратко рассмотрим каждый из них, приведя их обозначения в скобках:

Обычные файлы или regular files (-) —

наиболее общий тип файлов. Содержит данные в некотором формате — просто последовательность байт для операционной системы.

Директории или directories (d) —

файлы, содержащие имена находящихся в них файлов и указатели на метаданные, позволяющие ОС производить действия с этими файлами. Их удобно представлять в виде таблиц, в которых каждая строка содержит в себе жесткую ссылку.

Символьные ссылки или symbolic links (l) —

файлы, косвенно адресующие другие файлы файловой системы. Содержимое этих файлов интерпретируется как путь к файлу, на который они ссылаются.

Символьные устройства (c) и блочные устройства (b) —

два способа предоставить программам интерфейс для записи на какое-либо устройство. Канонически они различаются наличием или отсутствием буфера.

То есть в блочных устройствах (block devices) короткие порции данных склеиваются операционной системой в большие блоки, и обмен данными происходит уже этими самыми блоками. В символьных устройствах (character devices) обмен данными происходит посимвольно, то есть каждый символ будет отправляться сразу.

Именованные каналы или named pipes (p) —

файлы, используемые для передачи данных между процессами.

Сокеты или sockets (s) —

двусторонние каналы передачи данных между двумя единицами соединения.

В **Solaris OS** также присутствует **тип файлов Doors** (двери). Они имеют **обозначение D** (не путайте с d у директорий) и нужны для осуществления межпроцессного взаимодействия.

Глава 12

Файловый ввод-вывод

12.1 Открытие и закрытие файла

Как мы уже говорили, для того чтобы работать с файлом, нужно поместить информацию об этом файле в таблицу файловых дескрипторов. Для этой цели в ОС UNIX существует системный вызов **open (2)**.

open (2)

```
int open(const char *path,    /* path to file */
  int oflag,          /* access mode */
  /* mode_t mode */     /* used only on file creation */
);
```

В качестве аргументов этот системный вызов принимает путь к файлу, режим доступа и права доступа. Права доступа используются в случае, если мы вызываем open (2) с целью создания файла (в некоторых реализациях библиотеки Си, альтернативой является **creat (2)**). Системный вызов open (2) возвращает номер файлового дескриптора или код ошибки.

Приведем основные режимы доступа

Значение oflag	Назначение
O_RDONLY	только для чтения
O_WRONLY	только для записи
O_RDWR	для записи/чтения
O_CREAT	создать, если не существует
O_APPEND	запись с конца
O_TRUNC	запись с начала (фактически обрезать файл до нулевой длины)

Какие могут произойти ошибки при открытии файла?

При открытии файла могут произойти ошибки в случае, если мы не имеем доступа к файлу или не нашлось устройство, на котором этот файл располагается. Также ошибка может произойти, если в таблице файловых дескрипторов больше нет свободных строк.

Когда файл станет нам окончательно не нужен, можно закрыть его с помощью системного вызова close(2). Тем самым мы освободим одну строку в таблице файловых дескрипторов. Как вы помните, эта таблица имеет ограничения, поэтому это может быть важным. Возвращает 0 или код ошибки.

close(2)

```
int close(
  int fildes /* descriptor of file */
);
```

В каком случае close может вернуть код ошибки?

Единственный вариант — при передаче неверного файлового дескриптора. Т.е. этот файловый дескриптор, который не был открыт в таблице.

12.2 Перемещение внутри файла

Системный вызов lseek (2) позволяет оперировать с указателем внутри файла с помощью аргументов whence (действие с указателем) и offset (смещение позиции в байтах). Что позволяет производить дальнейшие операции чтения/записи с файлом, начиная с установленного lseek положения указателя. Этот системный вызов возвращает полученное смещение в байтах или код ошибки.

lseek

```
off_t lseek(
  int fildes,     /* opened file descriptor number */
  off_t offset,     /* position offset (bytes) */
  int whence      /* action to pointer */
);
```

Значения whence	Назначение
SEEK_CUR	Указатель сместится на offset относительно своей текущей позиции внутри файла
SEEK_SET	Указатель сместится на offset от начала файла
SEEK_END	Указатель сместится на offset от конца файла
SEEK_DATA	Перемещение указателя к началу следующей файловой дырки
SEEK_HOLE	Перемещение указателя к началу следующего сегмента данных

В каких случаях может не получиться сместить указатель файла?

Собственно, вариант того, что ОС откажется смещать указатель, только один — если это файл, который не поддерживает указатель внутри себя. К таким файлам относится FIFO, символьные устройства, сокеты и символьные ссылки.

12.3 Ввод-вывод

12.3.1 Простой ввод-вывод

Что нам нужно для простейшего ввода и вывода? Файловый дескриптор, полученный системным вызовом open (2), или дескриптор стандартного потока и буфер, который мы заранее подготовили для данных.

Также нам потребуется указать количество байт, которое мы хотим прочитать или записать.

Для чтения заданного количества байт используется системный вызов read (2), а для записи — системный вызов write (2).

Важно: Оба системных вызова будут смещать файловый указатель на указанное в nbyte количество байт.

read(2)

```
ssize_t read(
  int fildes,   /* number of opened file descriptor */
  void *buf,    /* reading buffer */
  size_t nbyte  /* byte quantity */
);
```

Возвращает количество прочитанных байт или код ошибки.

write(2)

```
ssize_t write(
  int fildes,      /* number of opened file descriptor */
  const void *buf,   /* writting buffer */
  size_t nbyte     /* byte quantity */
);
```

Возвращает количество записанных байт или код ошибки.

Помните, что read (2) и write (2) могут вернуть значение меньшее, чем nbyte. Это произойдет в случае, если до конца файла осталось меньше байт, чем мы запросили.

Пример чтения/записи.

Рассмотрим пример чтения/записи с помощью системных вызовов read (2) и write (2).

main.c

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

#define BUF_SIZE 256

int main(int argc, char *argv[]) {
  int bytes;
  char buf[BUF_SIZE];

  while((bytes = read(STDIN_FILENO, buf, BUF_SIZE)) > 0)
    if (write(STDERR_FILENO, buf, bytes) < 0)
      return EXIT_FAILURE;

  return bytes;
}
```

Алгоритм работы примерно следующий: сначала внутри функции main на стеке выделяется переменная bytes, которая будет хранить количество байт, и буфер размером 256.

Далее следует цикл, в котором условием продолжения является положительное количество считанных read(2) байт (Условие положительности покрывает и случай возникновение ошибки системного вызова, когда код возврата равен отрицательному числу). В данном случае мы читаем из стандартного потока ввода.

При записи write(2) считанных из STDIN байт в STDERR в условии if, мы также проверим не произошло ли ошибки в процессе работы системного вызова write.

В конце концов мы возвращаем bytes. Если у нас bytes = 0, т.е. read прочитал 0 байт, то фактически файл закончился (это признак конца файла). Если же bytes, какой-то отрицательный, то вернется ненулевой код возврата и мы увидим, что произошла ошибка.

12.3.2 Ввод-вывод со смещением

Мы уже написали о том, что есть простой ввод-вывод write и read, однако ему есть альтернатива — ввод-вывод со смещением.

Ввод-вывод со смещением позволяет не просто вводить и выводить в том место, на которое указывает текущая позиция внутри файла, а позволяет осуществить ввод в конкретную позицию или прочитать оттуда данные.

Для его осуществления есть два системных вызова pread (2) и pwrite (2). Оба вызова принимают в качестве аргуметнов файловый дескриптор, буфер для чтения/записи, размер этого буфера и смещение относительно начала файла.

pread (2)

```
ssize_t pread(
  int fildes,   /* descriptor of file */
  void *buf,    /* buffer */
  size_t nbyte,   /* size of buffer  */
  off_t offset  /* offset of start of file */
);
```

Возвращает количество считанных байт или код ошибки.

pwrite (2)

```
ssize_t pwrite(
  int fildes,     /* descriptor of file */
  const void *buf,  /* buffer */
  size_t nbyte,   /* size of buffer  */
  off_t offset    /* offset of start of file */
);
```

Возвращает количество записанных байт или код ошибки.

Когда может понадобиться ввод или вывод со смещением?

Если у вас есть однопоточное приложение, внутри открытого файла вы можете управлять указателем с помощью lseek. Если же у вас несколько потоков и они работают с одним и тем же файлом (например, большая база данных), а указатель все равно один, может произойти такая неприятная ситуация.

Допустим, вы поставили указатель на то место, куда хотели бы записать 0.5Гб, и начали запись. В это время другой поток что-то прочитал, то есть подвинул указатель. Теперь запись по этому указателю может привести к фатальным последствиям — испортятся важные данные.

Системные вызовы выполняются атомарно. То есть, при выполнении любого системного вызова блокируется выполнение всего процесса целиком и полностью — никто другой в этот момент не может сделать системный вызов из этого процесса. Это значит, что и pread и pwrite выполняются атомарно т.е. никто не сможет воспользоваться lseek и сдвинуть наш указатель. Именно этим и пользуются программисты, когда используют ввод и вывод со смещением.

12.3.3 Векторный ввод-вывод

Что делать, если нам нужно вывести сразу несколько буферов или прочитать сразу в несколько мест из одного файла? Это решается вводом дополнительного буфера. Это плохо тем, что это требует лишнего копирования данных внутри памяти. Для решения этой проблемы был придуман векторный (или рассеяныый) ввод вывод.

Одна из особенностей реализации векторного ввода-вывода — структура iovec. Она хранит в себе адрес начала сегмента (iov_base), в который мы хотим что-то записать, и количество байт (iov_len), которые мы готовы

отвести под хранение данных.

Структура iovec

```
typedef struct iovec {
  void *iov_base;   /* start address */
  size_t iov_len;   /* segment length */
} iovec_t;
```

Что нам это позволяет делать?

Теперь, создав массив из векторов iovec, мы можем последовательно брать вектор и читать/писать данные из блоков, которые в нем описаны. Эту процедуру за нас полностью выполняют системные вызовы readv (2) и writev (2). В качестве агрументов они принимают файловый дескриптор, указатель на первую структуру в массиве iovec и количество структур, содержащихся в массиве.

Системный вызов readv (2) отвечает за векторный вывод и возвращает количество считанных байт.

readv (2)

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

ssize_t readv(
  int fildes,         /* file descriptor */
  const struct iovec *iov,  /* pointer on first structure in array */
  int iovcnt          /* size of array of structres */
);
```

Системный вызов write (2) отвечает за векторный вввод и возвращает количество записанных байт.

writev

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

```
3
4  ssize_t writev(
5    int fildes,          /* file descriptor */
6    const struct iovec *iov,  /* pointer on first structure in array */
7    int iovcnt           /* size of array of structres */
8  );
```

Пример

```
1  ssize_t bytes_written;
2  int fd;
3  char *buf0 = "Inspiration unlocks the future.\n";
4  char *buf1 = "Now go, and don't look back.\n";
5  int iovcnt;
6  const struct iovec iov[2];
7
8  iov[0].iov_base = buf0;
9  iov[0].iov_len = strlen(buf0);
10 iov[1].iov_base = buf1;
11 iov[1].iov_len = strlen(buf1);
12
13 iovcnt = sizeof(iov) / sizeof(struct iovec);
14   ...
15 bytes_written = writev(fd, iov, iovcnt);
```

В этом примере в файл с дескриптором fd будут записаны две строки из buf0 и buf1.

12.3.4 Конец файла

Подробнее о том как работает конец файла. Никакого магического символа конца файла нет. У файла есть inode, а внутри него есть размер файла, т.е. сколько байт внутри этого файла.

Если же мы попытаемся прочитать большее количество байт ОС, это понимает и возвращает то количество байт, которое мы можем прочитать.

Причем мы помним, что внутри файла всегда есть указатель. Указатель всегда смещается при чтении или записи, и в какой-то момент нам вернутся оставшиеся к чтению байты, количество которых будет меньше размера буфера. Если мы попытаемся произвести чтение еще раз, ОС увидит, что указатель на конце файла, и вернет 0. Это будет являться признаком конца файла.

Что есть символ конца файла? Что вводит нас в заблуждение?

Мы знаем, что есть некий специальный символ, ввод которого внутрь терминала или внутрь эмулятора терминала, заставляет терминал сообщить ОС, что ввод закончен. То есть, если в терминале выполняется считывание с STDIN и мы что-то вводим, то последовательность ctrl + D даст ОС понять, что ввод (файл) закончен. Внутри файла этот символ не хранится нигде, это управляющий символ терминала для сообщения о конце ввода. Он никак не говорит о том, что файл закончился.

Глава 13

Директории

С точки зрения пользователя файлы в ОС UNIX организованы в виде древовидного пространства имен. Дерево состоит из ветвей (каталогов) начиная от (/) и заканчивается листьями (файлами).

Каталог отличается от обычного файла тем, что у него четко заранее заданная известная структура. Которую можно представить в виде таблицы содержащей имена находящихся в нем файлов и указатели на метаданные, позволяющие ОС производить действия с этими файлами.

13.1 Структура dirent

По своей сути, директория является массивом структур dirent. Эта структура характеризует только жесткие ссылки внутри директории. Первое поле структуры dirent — номер индексного дескриптора, второе поле — имя файла жесткой ссылки.

Структура dirent

```
1  typedef struct dirent {
```

```
    ino_t d_ino;       /* inode number */
    char d_name[];       /* name of file */
  } dirent_t;
```

В разных ОС структура dirent имеет различное описание в заголовочных файлах — иногда в ней есть другие поля.

13.2 Получение текущей рабочей директории и ее изменение

Для изменения рабочего каталога существует следующий набор системных вызовов: chdir (2) принимает в качестве единственного аргумента путь к файлу, а fchdir (2) — файловый дескриптор. Оба системных вызова возвращают 0 или код ошибки.

chdirr (2)

```
  int chdir(
    const char *path
  );
```

fchdirr (2)

```
  int fchdir(
    int fildes
  );
```

Для получения пути к текущей рабочей директории есть системный вызов getcwd (2). Он возвращает указатель на буфер, в котором хранится текущий рабочий каталог, или код ошибки.

getcwdr (2)

```
  char * getcwd(
    char *buf,
```

```
    size_t size
);
```

13.3 Создание и удаление директорий

Для создания директорий есть системный вызов mkdir (2). Он принимает путь к новой директории и режим доступа mode.

mkdir(2)

```
int mkdir(
    const char *path, /* path to directory */
    mode_t mode      /* access mode */
);
```

mode_t – это режим доступа, который будет установлен на директорию. Этот режим доступа складывается с umask и маской родительской директории.

Системный вызов rmdir (2) принимает путь и удаляет директорию в случае, если директория имеет всего две жестких ссылки, это точка и две точки. Если внутри директории больше жестких ссылок, то директория не удалится, поскольку она не пустая с точки зрения ОС.

rmdir(2)

```
int rmdir(
    const char *path /* path to directory */
);
```

Оба вызова возвращают 0 или код ошибки.

Глава 14

Ссылки

14.1 Жесткие ссылки

Когда мы говорим о ссылках, чаще всего мы говорим о жестких ссылках. Важно понимать, что внутри каталога могут быть две и более жестких ссылок, ссылающихся на один и тот же inode. Ссылка — это логическое понятие, inode - физическое. Таким образом, мы можем говорить, что файл доступен по двум путям.

Еще раз. Имя не является атрибутом файла, оно является алиасом, по которому мы можем получить номер индексного дескриптора. (Читайте про структуру dirent)

Системный вызов link (2) создает жесткую ссылку на файл. В качестве аргументов link (2) принимает путь к файлу, на которой мы хотим создать ссылку, и, соответственно, путь для этой самой ссылки.

link(2)

```
1  int link(
2    const char *existing, /* path to file */
```

```
    const char *new      /* path to link */
  );
```

Возвращает: 0 или код ошибки.

Важно: Жесткую ссылку возможно создать только в пределах одной файловой системы.

Системный вызов unlink (2) позволяет удалить жесткую ссылку на файл. В качестве аргумента unlink (2) принимает путь к файлу, по которому мы хотим удалить ссылку. Возвращает 0 или код ошибки.

unlink (2)

```
  int unlink(
    const char *path
  );
```

Создание и удаление жесткой ссылки состоит из двух операций:

- модификация каталога. Для этого у нас должны быть права на запись в каталог;
- модификация inode, который описывает этот файл — мы должны увеличить на 1 nlink при создании жесткой ссылки и уменьшить на 1 nlink при удалении.

Вопрос: что сделать первым? Сначала мы должны изменить директорию. Чтобы у нас не получилось ситуации, когда жесткая ссылка в каталоге ссылается на файл, которого нет.

Важно: Файл будет являться удаленным, когда на него не останется ни одной жесткой ссылки (nlink = 0).

14.2 Символьные ссылки

Наряду с жесткими ссылками есть такое понятие, как символьные ссылки. Это абсолютно разные вещи. Жесткая ссылка — это запись внутри каталога. Символьная ссылка — это отдельный файл со своей собственной inode, с собственными блоками данных и типом symlink. Внутри символьной ссылки содержится некая строчка (абсолютно любая, какую запишем). Эту строчку при работе с символьной ссылкой ядро ОС будет считать путем, по которому нужно пройти к основному файлу.

Для создания символьной ссылки существует системный вызов symlink (2). Его аргументами являются путь к существующему файлу и путь к файлу-символьной ссылке, которую мы хотим создать.

symlink(2)

```
#include <unistd.h>

int symlink(
  const char *name1,
  const char *name2
);
```

Для чтения содержимого символьной ссылки существует readlink (2). Его аргументы — путь к символьной ссылке, буфер для хранения прочитанного и размер буфера (количество байт, которые мы хотим прочитать).

readlink(2)

```
#include <unistd.h>

ssize_t readlink(
  const char *restrict path,  /* path to link */
  char *restrict buf,     /* buffer */
  size_t bufsiz       /* size of buffer */
);
```

Глава 15

Файлы устройств

Файлы устройств обычно предоставляют интерфейс для доступа к принтерам и прочим периферийным устройствам, но они также могут быть использованы для доступа к ресурсам на этих устройствах, например, разделам диска. Также они могут быть использованы для эмуляции доступа к системным ресурсам, которые не подключены к реальным устройствам, например, генератору псевдо-случайных чисел (/dev/urandom).

Для создания файлов блочных и символьных устройств можно использовать системный вызов mknod (2). Он позволяет сделать жесткую ссылку внутри файловой системы на файл, который будет сам по себе ссылаться на определенное устройство, используя механизм ссылки на устройство. Он принимает путь к файлу, желаемый режим досупа и непосредственно структура устройства.

mknod (2)

```
int mknod(
  const char *path, /* path to file */
  mode_t mode,    /* access mode */
  dev_t dev       /* device */
);
```

Важно: На самом деле этот системный вызов можно использовать и для создания файлов и именованных каналов.

Часть V

Пользователи и группы

Глава 16

Пользователи и группы

Как мы помним, UNIX является многопользовательской операционной системой. Пользователи, занимающиеся общими задачами, могут объединяться в группы. Каждый пользователь обязательно принадлежит к одной или нескольким группам. Все команды выполняются от имени определенного пользователя, принадлежащего в момент выполнения к определенной группе.

Для изменения действующего пользователя существует улилита su.

16.1 Атрибуты пользователя

В ОС UNIX пользователь представляет собой некоторую сущность, которая обладает некими атрибутами. Атрибуты каждого пользователя можно увидеть в файле /etc/passwd. Этот файл хранит информацию о пользователях системы.

Приведем список атрибутов пользователя.

name : password : UID : GID : GECOS: home : shell

- **name** —

  логин;

- **x** —

  пароль (сейчас перемещен например в /etc/master.passwd (FreeBSD) или /etc/shadow);

- **UID** —

  идентификатор пользователя (обычно от 0 до 65535);

- **GID** —

  идентификатор главной группы;

- **GECOS** —

  вспомогательная информация о пользователе (например, имя, номер телефона, адрес).
  Для изменения GECOS используется утилита chnf;

- **home** —

  домашний каталог пользователя;

- **shell** —

  командный интерпретатор, запускаемый по умолчанию.

16.1.1 Атрибуты группы

Файл /etc/group хранит информацию о всех группах системы.

Приведем список атрибутов гррупы.

name : x : GID : users

- **name** —

  символьное имя;

- **x** —

  устаревшее поле (раньше был пароль);

- **GID** —

  идентификатор группы;

- **users** —

  список имен пользователей, для которых эта группа не является главной, разделенных запятыми.
  Важно: Если вам нужно получить главную группу пользователя, обращайтесь к файлу /etc/passwd.

16.1.2 Утилита getent

Если вам требуется найти или отсортировать пользователей или группы, то вам нужна база пользователей или групп. Эти базы собираются из нескольких мест. Чтобы обратится к ним, можно воспользоваться утилитой **getent** (получить записи из базы). В качестве аргумента указывается имя базы, например, passwd.

Также можно воспользоваться ключом. Так в passwd ключом будет имя пользователя.

Например

```
1  ag@helios:/home/ag$ getent passwd good_user
```

Выведет информацию о пользователе good_user, если такой имеется в системе.

Часть VI

Владельцы файлов, права и режимы доступа

Глава 17

Права и режимы доступа

У файла есть три группы модификаторов:

- права владельца — обозначаются u (user);
- права для членов группы владельца — обозначаются g (group);
- права для всех остальных — обозначаются o (other).

Пользовательские права доминируют над групповыми. Если кто-то другой принадлежит группе, владеющей файлом, то права группы рассматриваются с большим приоритетом, чем права для остальных.

Модификаторы доступа к файлам.

Существуют три основных модификатора доступа: чтение (r — read), запись (w — write) и выполнение (x - eXecute). Установленный модификатор дает пользователю, группе или остальным соответствующее этому модификатору право.

Модификаторы доступа для файлов и жестких ссылок дают возможность:

- **r** — просматривать содержимое;
- **w** — записывать/редактировать файл;
- **x** — попытаться выполнить как программу.

Модификаторы доступа для каталога дают возможность:

- **r** — просматривать список содержащихся файлов;
- **w** — создавать файлы и каталоги внутри этого каталога;
- **x** — войти в каталог.

Для символических ссылок права определяются правами файла, на которой они указывают.

Также существуют три специальных бита SUID, SGID и Sticky bit:

- **SUID (Set User ID)** — позволяет пользователю запустить исполняемый файл от имени настоящего владельца файла (с правами владельца);
- **SGID (Set Group ID)** — позволяет пользователю запустить исполняемый файл от имени группы настоящего владельца файла (с правами группы);
- **Sticky bit** — разрешает всем пользователям писать в файл или каталог, но право удаления остается только за владельцем.

Списки контроля доступа.

ACL (Access Control Lists) —

расширенные списки контроля доступа.

Это дополнительная информация о файлах, которая позволяет конкретным отдельным пользователям или группам задать другие своеобразные

специфические права.

17.1 Права доступа

17.1.1 Проверка прав доступа

Проверить права доступа к файлу можно с помощью системного вызова access (2). Его аргументы интуитивно понятны — путь к файлу path и режим доступа amode.

access (2)

```
int access(
  const char *path,
  int amode
);
```

В amode можно указать следующие режимы:

R_OK чтение;

W_OK запись;

X_OK исполнение;

F_OK существование.

Системной вызов access (2) возвращает 0 в случае, если у файла установлены заданные в <amode> права, и код ошибки в ином случае.

Также режимы можно склеивать через операцию дизъюнкции (|).

Например

```
access“(/home/ag/“myfile , W_OK | X_OK);
```

Вернет код успеха, в случае если файл “/home/ag/myfile“ доступен на запись и выполнение.

17.1.2 Изменение прав доступа

У файлов можно менять доступ, и для этого у нас есть два системных вызова: chmod (2) и fchmod (2). Отличаются они первым аргументом. В первом случае, это имя файла, у которого мы хотим поменять права, во втором случае, это файловый дескриптор открытого файла. Второй аргумент одинаковый и представляет из себя четырехразрядное восьмеричное число — желаемый режим доступа.

chmod (2)

```
int chmod(
  const char *path,
  mode_t mode
);
```

fchmod (2)

```
int fchmod(
  int fildes,
  mode_t mode
);
```

В mode можно указать следующие режимы: права на чтение, запись и исполнение для пользователя, группы и “остальных“, так и SUID, SGID и sticky bit.

Например

```
int chmod("/home/ag/myfile", S_ISUID | S_IWGRP);
```

Установит SUID и даст группе право на запись.

17.1.3 Маска создания файла

Мы помним, что существует маска создания файла, которая накладывается на тот режим, который указан третьим аргументов в системном вызове open. Этот режим будет складываться логическим оператором И с маской создания файла.

Для изменения маски создания файлов существует системный вызов umask (2), единственным аргументом принимающий значение маски. Этот вызов возвращает предыдущее значение маски.

umask (2)

```
mode_t umask(
  mode_t cmask /* mask value */
);
```

Глава 18

Владельцы файлов

18.1 Изменение владельца файла

С помощью системных вызов chown (2) можно изменить владельца файла. Оба системных вызова принимают в качестве второго и третьего аргументов идентификатор владельца и идентификатор группы. Единственная разница между ними — chown (2) принимает первым аргументом путь к файлу, а fchown (2) — файловый дескриптор.

chown(2)

```
int chown(
  const char *path,
  uid_t owner,
  gid_t group
);
```

fchown(2)

```
int fchown(
  int fildes,
  uid_t owner,
  gid_t group
```

```
);
```

Оба системных вызова возвращают 0 в случае успеха, в ином случае — код ошибки.

Часть VII

Память

Глава 19

Как устроена память

На рисунке изображена упрощенная концептуальная модель того как распределяется наше виртуальное адресное пространство между тем что будет в этом адресном пространстве существовать.

По младшим адресам оперативной памяти, как мы уже говорили, располагается код ядра, за ним следует стек. Стек растет к младшим адресам. Положить данные на вершину стека можно с помощью операции push и забрать с вершины с помощью операции pop.

Дальше идет неиспользованное, свободное по умолчанию пространство, называемое кучей. Куча — пространство между стеком и данными.

Коду приложения отводятся самые старшие адреса. Это пространство нужно для того, чтобы мы не могли случайно или специально поменять наш код и

заставить это приложение работать по-другому.

19.1 Аллокация памяти

Выделение памяти с помощью расширения сегмента данных

Как уже было сказанно ранее, при запуске процесса ОС выделяет процессу память для размещения кучи, оставляя между кучей и стеком свободное пространство “невыделенной” памяти. При необходимости процесс может запросить дополнительную память, необходимую ему для работы.

Дополнительная память выделяется начиная от старших адресов кучи. При этом указатель верхей границы смещается, тем самым отмечая, что по запрошенным адресам места больше нет.

Смещение границы кучи делается при помощи системных вызовов brk (2) и sbrk (2), которые смещают верхнюю границу кучи, изменяя количество выделенной процессу оперативной памяти. Системные вызовы называются так от слова break, так называется сам указатель — break pointer.

Системный вызов brk (2) устанавливает верхнюю границу кучи равной аргументу вызова.

brk (2)

```
#include <unistd.h>

int brk(
  void *endds
);
```

Системный вызов sbrk (2) передвигает верхнюю границу на количество байт, указанное при вызове. Значение может быть и отрицательным, в этом случае

размер кучи уменьшится.

sbrk (2)

```
#include <unistd.h>

void * sbrk(
  intptr_t incr
);
```

Выделение новых сегментов из анонимной памяти

Существует также такое понятие как анонимная память.

Анонимная память —

> область виртуальной памяти, которая не сопоставлена ни с каким файлом.

Например, у нас есть программа bin.sh, и образ этой программы хранится на диске. Когда мы его загружаем (копируем в оперативную память), он копируется в неанонимную память, потому что образ на диске есть. Если же мы возьмем на куче какой-то кусок и будем им пользоваться, то этот кусок кучи не будет связан с тем, что лежит на системе хранения данных, и это будет называться анонимной памятью.

Мы можем выделять кусочки анонимной памяти для своего собственного использования. Для этого существует системный вызов mmap (2).

mmap (2)

```
#include <sys/mman.h>

void *mmap(
  void *addr,
```

```
    size_t len,
    int prot,
    int flags,
    int fildes,
    off_t off
);
```

Перечислим аргументы mmap:

- addr — желаемый адрес (если установлен 0 или этот адрес не доступен, то система сама выберет адрес);
- len — количество отображаемых байт;
- prot — разрешающие права доступа к памяти;
  - PROT_READ;
  - PROT_WRITE;
  - PROT_EXEC;
  - PROT_NONE.
- flags - специальные флаги
  - MAP_SHARED — могут использоваться несколько процессов;
  - MAP_PRIVATE — только для вызывающего процесса;
  - MAP_FIXED — выделить память точно с addr.
- filedes — дескриптор файла;
- off — смещение отображенного участка от начала файла.

Снять отображение сегмента можно с помощью системного вызова munmap (2). Этот системный вызов принимает адрес начала сегмента и длину сегмента соответственно.

munmap (2)

```
int munmap(
  void *start,
  size_t length
);
```

Важно: Канонически mmap (2) нужен для того, чтобы отобразить файл в память.

Отображение файла в память —

способ работы с файлами, при котором файлу ставится в соответствие участок виртуальной памяти. При этом чтение данных из этих адресов фактически приводит к чтению данных из отображенного файла, а запись данных по этим адресам приводит к записи этих данных в файл.

Часть VIII

Процессы и потоки

Глава 20

Процессы

Процесс —

> совокупность программы и метаинформации, описывающей её выполнение

Когда мы говорим процесс, речь идет именно о выполняющейся программе. Когда программа не выполняется её нельзя назвать процесом.

Каждый процесс представлен в системе двумя основными структурами данных — proc и u_block, описанными, соответственно, в файлах <sys/proc.h> и <sys/user.h>. Содержимое и формат этих структур различны для разных версий UNIX.

20.1 Атрибуты процесса

К атрибутам процесса относятся:

- **PID** —

идентификатор процесса.

- **PPID** —

  идентификатор родительского процесса.

- **UID** —

  идентификатор пользователя - владельца процесса.

- **GID** —

  идентификатор группы - владельца процесса.

- **Nice Number** —

  приоритет процесса.

- **TTY** —

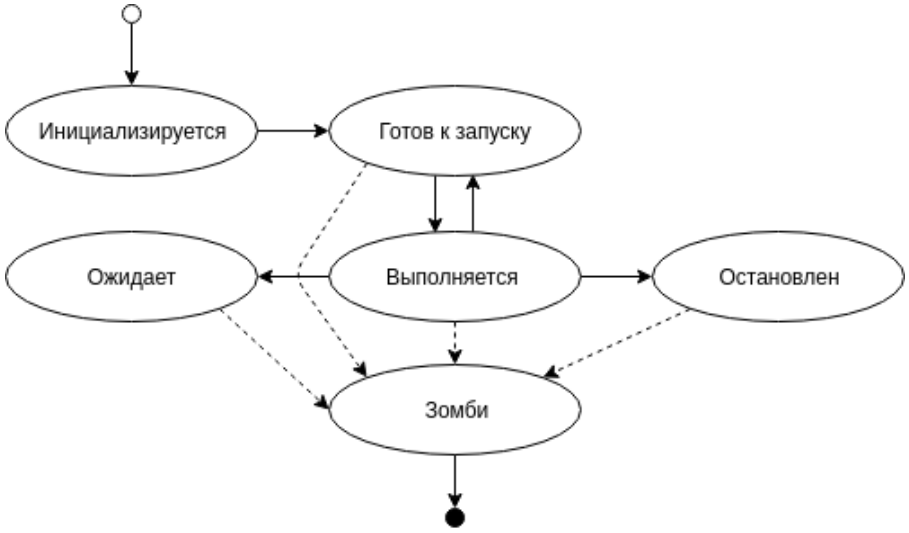
  терминальная линия.

20.2 Жизненный цикл процесса

Жизненный цикл процесса начинается с того, что его кто-то порождает. Единственный процесс, который никто не порождает, это процесс init. Порождение процесса осуществляется с помощью fork (2) или vfork (2).

Первая стадия в жизни процесса — **“инициализация“**. В это время ядро производит подготовительные работы к дальнейшей работе процесса. На самом деле, это не совсем состояние, но логически для пользователя оно

существует.



Когда инициализация процесса завершается — он оказывается в стадии **“готов“**. С этого момента процесс находится в ожидании момента, когда его выберет планировщик процессов и даст ему какой-то квант процессорного времени.

При получении процессорного времени процесс переходит в состояние **“выполняется“**. Выполняться процесс может в режиме ядра — при осуществлении системных вызовов, прерываний и в режиме задачи — выполнять инструкции процессора. По окончании квоты времени, процесс может снова вернуться в состояние **“готов“**.

Также из состояния “выполняется“ процесс может перейти еще в два состояния — **“остановлен“** и **“ожидает“**. Если он остановлен, то он остановлен пользователем (например, получен сигнал SIGSTOP). Ожидание наступает тогда, когда процессу нужны какие-то ресурсы. Как только условие, которого ждет процессор, выполняется, он переходит в состояние **“готов“**, то есть ожидает своей очереди на выполнение.

Из каждого состояния процесс может перейти в в состояние **“зомби“**. Этот процесс нужен для того, чтобы родительский процесс мог получить код

возврата этого процесса. Это промежуточное состояние — процесса уже нет, но он технически еще есть. Тем не менее, это самое что ни на есть валидное состояние.

В какой момент процесс окончательно исчезает из таблицы процессов?

После того как его родительский процесс вызовет wait (2), waitpid (2), waitid(2) - позволяет завершить процесс, если он таки стал зомби.

Традиционная проблема — порождая процесс, необходимо в какой-то момент сказать ему wait (2), чтобы не плодить зомби. В противном случае у вас может закончится лимит на создание новых процессов.

Что происходит с зомби, если его родитель не вызвал wait (2), а погиб?

Его родителем становится init. Для всех процессов, которые становятся его потомками он делает wait (2). По факту, если вы написали какой-то код, который выполняет некоторое количество процессов после чего завершился, то на самом деле зомби в системе не останутся, т.к. после того как ваш основной процесс завершился, то потомков этого процесса подхватит init и он сам вызовет для них wait (2).

20.3 Создание процессов

20.3.1 Семейство системных вызовов fork

Как мы помним, все процессы кроме init порождаются системными вызовами fork(2) или vfork(2).

Системный вызов fork (2) создает дочерний процесс - точную копию родительского процесса. При успешном выполнении процессу родителю возвращается PID потомка, а потомку - 0.

fork (2)

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

pid_t fork(void);
```

Пример

```
int main(void) {
  pid_t pid;

  pid = fork(); /* change to vfork */

    if (pid == 0) {
        /* Now we're in child process */
  }
  else {
    /* Now we're in parent process */
  }
  return EXIT_SUCCESS;
}
```

Системный вызов vfork (2) определяется как fork (2) со следующим ограничением: поведение функции не определено, если созданный с её помощью процесс совершит хотя бы одно из следующих действий:

- Произведет возврат из функции, в которой был вызван vfork (2);
- Вызовет любую функцию кроме _exit() или exec* (2);
- Изменит любые данные кроме переменной, в которой хранится возвращаемое функцией vfork (2) значение.

vfork (2)

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

pid_t vfork(void);
```

Разница между fork (2) и vfork (2).

В отличие от fork (2), vfork (2) не создает копию родительского процесса, а создает разделяемое с родительским процессом адресное пространство до тех пор, пока не будет вызвана функция _exit или одна из функций exec. Родительский процесс на это время останавливает свое выполнение. Отсюда следуют и все ограничения на использование – дочерний процесс не может изменять никакие глобальные переменные или даже общие.

Другими словами, после вызова vfork() оба процесса будут видеть одни и те же данные и переменные.

Единственный вариант использования vfork — когда вы предполагаете, что в скором будущем вы выполните функцию exec, которая заместит ваше адресное пространство новым исполняемым файлом. Соответственно, начиная с этого момента, будет разблокировано выполнение процесса родителя.

Пример

```c
int main(void) {
  pid_t pid;
  int status, common_var = 0;

  pid = fork(); /* change to vfork */

    if (pid == 0) {
        /* Child  */
    common_var = 1;
    printf("Child common_var value: %i \n",  common_var);
```

```
11
12        exit(EXIT_SUCCESS);
13      }
14      waitpid(pid, &status, 0);
15      printf("Parent common_var value: %i \n",  common_var);
16
17      if (common_var) {
18        puts("vfork(): common variable has been changed");
19      } else {
20        puts("fork(): common variable hasn't been changed");
21      }
22      return EXIT_SUCCESS;
23    }
```

Вывод для fork (2)

```
1  Child common_var value: 1
2  Parent common_var value: 0
3  fork(): common variable hasn't been changed
```

Вывод для vfork (2)

```
1  Child common_var value: 1
2  Parent common_var value: 1
3  vfork(): common variable has been changed
```

20.3.2 Семейство системных вызовов exec

Мы уже сказали, что использование системного вызова vfork (2) рационально в случае дальнейшего запуска функции из семейства exec (3). Функция exec (3) загружает и запускает программу, заданную в аргументе path. Запущенная программа замещает адресное пространство текущего процесса. Все файлы вызывающей программы остаются открытыми и являются доступными новой программе.

Программа должна быть или двоичным исполняемым файлом, или скриптом, начинающимся со строки вида “#! интерпретатор [аргументы]“.

execle(3)

```c
#include <unistd.h>

int execle(
  const char *path,   /* path to file */
  const char *arg0,   /* args */ /* ... */,
  (char *)0,      /* NULL */
  char *const envp[]  /* environment */
);
```

Семейство функций exec (По сути модификации execle) execlp() , execvp() , execl(), execv() , execle() , execve()

Значения суффиксов l, v, p и e:

p поиск программы по путям переменной окружения PATH. Без суффикса поиск будет производиться только в рабочем каталоге.

l показывает, что адресные указатели (arg0, arg1, ..., argn) передаются, как отдельные аргументы.

v показывает, что адресные указатели (arg[0], arg[1],...arg[n]) передаются, как массив указателей.

e показывает, что “дочернему“ процессу может быть передан аргумент envp (массив с переменными окружения).

Глава 21

Межпроцессное взаимодействие

Далеко не всегда поставленную задачу можно решить используя только один процесс. В UNIX ОС существует некоторое количество механизмов для межпроцессного взаимодействия.

Перечислим их в произвольном порядке:

- System V семафоры;
- каналы;
- сокеты;
- сигналы;
- очереди сообщений;
- System V разделяемая память;
- механизм STREAMS.

21.1 Семафоры

Семафор —

> это объект синхронизации, имеющий множество состояний: 0(заблокирован), 1, 2, . . . , N

Семафор представляет из себя счетчики, ограничивающий количество возможных одновременных пользователей ресурса. Значение семафора отражает количество свободных мест.

Семафоры находятся в адресном пространстве ядра, поскольку доступ к ним должны иметь все процессы.

Примерный алгоритм работы с семафором

Изначально семафору устанавливается целое положительное значение N — максимальное значение одновременных пользователей какого-либо ресурса.

Когда новый пользователь хочет захватить (начать работать с ресурсом), он уменьшает значение семафора на 1. Таким образом уже N-1 пользователей могут начать использовать ресурс в данный момент. После окончания работы с ресурсом, пользователь должен увеличить значение N на 1, тем самым освободив место для еще одного пользователя.

Если значение N становится равным 0, это значит, что уже никто не может получить доступ к желаемому ресурсу.

Существуют POSIX и System V семафоры. Семафоры System V являются не отдельными счетчиками, а представляют из себя группу счетчиков, объединенных каким-либо признаком.

Для работы с семафорами System V используются такие системные вызовы как:

```c
#include <sys/sem.h>

int semctl(int semid, int semnum, int cmd, ...);

int semget(key_t key, int nsems, int semflg);

int semids(int *buf, uint_t nids, uint_t *pnids);

int semop(int semid, struct sembuf *sops, size_t nsops);

int semtimedop(int semid, struct sembuf *sops, unsigned nsops,
    struct timespec *timeout);
```

Для работы с семафорами POSIX используются такие функции как:

```c
#include <semaphore.h>

int sem_wait(sem_t *sem);

int sem_init(sem_t *sem, int pshared, unsigned int value);

int sem_post(sem_t *sem);

int sem_getvalue(sem_t *sem, int *sval);

int sem_destroy(sem_t *sem);
```

Вам предлагается ознакомиться с ними самостоятельно.

21.2 Каналы

21.2.1 Неименованные каналы

Для создания неименованного калала существет системные вызовы pipe (2) и pipe2 (2). Оба они принимают на вход массив из двух элементов. После успешного выполнения вызова массив содержит два файловых дескриптора: для чтения информации из канала и для записи в него соответственно.

pipe(2)

```
#include <unistd.h>

int pipe(
  filedes[2]
);
```

Системный вызов pipe2 (2) принимает также некоторые флаги, влияющие на поведение канала.

Неименованные каналы можно использовать для родственных процессов. Когда процесс порождает другой процесс, дескрипторы родителя наследуются дочерним процессом, и, таким образом, осуществляется связь между двумя процессами. Один из них использует канал только для чтения, а другой только — для записи.

21.2.2 Именованные каналы

Именованные каналы во многом работают так же, как и неименованные каналы, но все же имеют несколько заметных отличий.

- именованные каналы существуют в виде специального файла устройства

в файловой системе;

- процессы различного происхождения могут разделять данные через такой канал;
- именованный канал остается в файловой системе для дальнейшего использования и после того, как весь ввод/вывод сделан.

Существует два способа создания именованного канала:

Создать обычный файл, директорию или файл специального назначения с помощью системного вызова mknode (2), указав 0 в dev_t.

mknode (2)

```
int mknod(
  const char *path,
  mode_t mode,
  dev_t dev
);
```

или воспользоваться функцией mkfifo (3)

mkfifo (3)

```
int mkfifo(
  const char *pathname,
  mode_t mode
);
```

21.3 Сокеты

Как уже было сказано, сокеты — это двусторонние каналы передачи данных между двумя единицами соединения. Существует несколько видов сокетов:

- UNIX domain

- Сетевые
- BSD сокеты
- Прочие

Они различаются по семействам адресов.

Сокет подразумевает, что у вас есть некий сервер, который слушает определенный сокет.

Для работы с сокетами существуют следующие системные вызовы.

Системный вызов socket(2) создает новый сокет указанного типа и возвращает номер файлового дескриптора или код ошибки. В качестве аргументов этот системный вызов принимает семейство адресов, тип соединения и протокол.

socket(2)

```
int socket(
  int domain,   /* address domain */
  int type,   /* type */
  int protocol  /* protocol (ex. tcp, udp) */
);
```

Системный вызов bind(2) используется на стороне сервера для того, чтобы ассоциировать сокет с адресом (описанным структурой address), например, адресом хоста и портом. Этот системный вызов возвращает 0 или код ошибки. Вызов принимает номер файлового дескриптора сокета, имя и длину поля имени.

bind(2)

```
int bind(
  int s,              /* socket's descriptor number */
  const struct sockaddr *name,  /* name */
  int namelen           /* size of name in bytes */
```

```
);
```

Системный вызов listen(2) используется на стороне сервера для того, чтобы TCP сокет, связанный с адресом, начал “слушать” указанный порт. Системный вызов принимает номер файлового дескриптора сокета и длину очереди. Возвращает 0 или код ошибки.

listen(2)

```
int listen(
  int s,      /* socket's descriptor number */
  int backlog   /* length of queue */
);
```

Системный вызов accept(2) спользуется на стороне сервера. Принимает входящее соединение, создает новое TCP соединение и новый сокет, связанный с подключенным. Возвращает номер дескриптора или код ошибки.

accept(2)

```
int accept(
  int s,          /* socket's descriptor number */
  struct sockaddr *addr,  /* client's address */
  socklen_t *addrlen    /* size of client's address */
);
```

Системный вызов connect(2) используется на стороне клиента и назначает свободный порт сокету. В случае TCP сокета пытается установить новое TCP соединение. Этот вызов принимает номер файлового дескриптора сокета, имя и длину поля имени. Он возвращает 0 или код ошибки.

connect(2)

```
int connect(
  int s,              /* socket's descriptor number */
  const struct sockaddr *name,  /* name */
  int namelen           /* size of name in bytes */
);
```

21.4 Сигналы

Сигнал —

механизм ОС для уведомления процесса о некотором событии.

Идея в том, что когда-то возникла необходимость асинхронно отправлять уведомления другому процессу о событии (как бы просто ставить галочку о том, что событие произошло). Для этой задачи и придумали механизм сигналов.

В случае только межпроцессного взаимодействия ОС не гарантирует доставку сигналов.

Есть определенное количество сигналов и, в зависимости от операционной системы, это количество меняется. Используя сигналы, нужно понимать, что это может сделать ваш код системно зависимым — непереносимым.

Стандартных сигналов 16, но в большинстве ОС их немного больше. Они говорят о том, что нужно делать когда приходит сигнал такого типа. Вот некоторые из таких реакций:

SIG_IGN проигнорировать пришедший сигнал.

SIG_INT завершить процесс.

SIG_ERR завершить процесс с дампом памяти.

SIG_HOLD позволяет остановить процесс.

Существуют некоторые немаскируемые сигналы. В первую очередь, это сигнал **SIG_KILL**. Дословно — убить процесс. Однако, существует несколько случаев, при которых после вызова SIG_KILL сигнал не будет убит сразу же:

- когда процесс является зомби;
- когда процесс остановлен. Вы можете остановить выполнение процесса, но после того как вы его запустите он все же умрет;
- когда процесс находится в вызове системного вызова. До тех пор пока системный вызов не будет завершен сигнал не будет обработан

Легкий способ сделать неубиваемый процесс — взять подключиться по сети к какой-нибудь файловой системе, открыть там файл и начать его читать, а потом физически выключить сервер. Вызов read (2) будет пытаться достучаться до сервера, который не будет отвечать. Убить такой процесс можно только двумя способами: включить сервер или перезагрузить систему.

SIG_INT прерывание программы, которое обычно посылает эмулятор, поймавший обработку нажатия CTRL+C.

Сигнал **SIG_TERM** посылается, когда вы совершаете команду kill PID. Это еще один из тех сигналов, которые нельзя перехватить.

SIG_STOP останавливает процесс. Посылается после нажатия CTRL+Z всем процессам текущей группы.

SIG_CONT продолжить выполнение, остановленного процесса

SIG_ALARM вы можете воспользоваться этим сигналом в случае, если вы хотите уведомить сами себя через какое-то время, что ваше время истекло.

SIG_PIPE вы можете получить данный сигнал, когда ваш сокет был неожиданно завершен второй стороной по какой-либо причине.

Сигналы SIG_USR1 и SIG_USR2 — два стандартных сигнала, которые по умолчанию приводят к завершению процесса. Это такой способ послать уведомление процессу о том, что произошло что-то не предусмотренное стандартным поведением системы.

21.4.1 Обработка сигналов

На все сигналы, кроме SIG_STOP и SIG_KILL, вы можете самостоятельно написать свой обработчик сигналов — собственную функцию, которая будет реагировать на пришедший сигнал так, как вы в ней описали.

Свой обработчик сигналов можно сделать разными способами.

Библиотека libc предлагает механизм, который называется signal (3). Эта функция принимает в качестве аргементов номер сигнала и новую диспозицию сигнала. В качестве диспозиции можно подать функцию, которую вы хотите назначить обработчиком сигнала. Функция возвращает предыдущую диспозицию.

signal (3)

```
#include <signal.h>

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

Какие недостатки есть у signal (3)?

Во многих реализация libC диспозиция сигнала устанавливается на действие по умолчанию каждый раз при получении сигнала.

Кроме того, у процесса в контексте есть такое понятие, как маска принимаемых сигналов. Фактически, это битовая последовательность, которая маскирует сигналы от принятия при их передаче. Если в этой битовой последовательности первый бит установлен в единицу, то, при попытке доставить сигнал процессу, операционная система увидит, что этот сигнал замаскирован, и поэтому его доставлять не нужно — он будет просто отброшен.

Есть еще один более гибкий традиционно используемый способ обработки сигналов — это системный вызов sigaction (2), который принимает числовой номер сигнала и два указателя на структуру sig_action.

Первый указатель нужен для того, чтобы указать ОС, что мы хотим делать в случае прихода этого сигнала т.е. Во второй указатель помещается реакция на сигнал после выполнения системного вызова sigaction (2).

sigaction (2)

```
#include <signal.h>

    int sigaction(int sig, const struct sigaction *restrict act,
```

```
4         struct sigaction *restrict oact);
```

Структура sigaction включает в себя следующие поля:

- void (*sa_handler)(int) – указатель на функцию обработчик сигнала;
- void (*sa_sigaction)(int, siginfo_t *, void *) – указатель на функцию обработчик сигнала, если установлен флаг SA_SIGINFO;
- sa_flags – флаги;
- sa_mask – битовая маска, которая маскирует прием/доставку сигналов процессу.

21.5 Очереди сообщений

Очередь сообщений —

> механизм передачи некоторого объема данных между процессами, основанный на сигналах

Очередь сообщений используется, когда вам нужно передать какому-то клиенту или блоку информацию о том, что необходимо обработать. Позволяет передавать небольшие объемы данных между процессами.

Очередь могут использовать все процессы, имеющие ключ и обладающие правами доступа к очереди (msg_perm).

Структура сообщения очереди имеет следующие поля:

- тип;
- длина сообщения в байтах;

- тело сообщения.

Для реализации очередей используются такие системные вызовы как:

```c
#include <sys/msg.h>

int msgctl(int  msqid, int cmd, struct msqid_ds *buf);
int msgget(key_t key, int msgflg);
int msgsnd(int  msqid, const void *msgp, size_t msgsz, int msgflg);
```

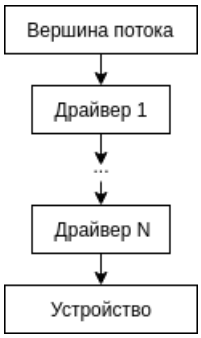
21.6 Разделяемая память

21.7 Механизм STREAMS

Механизм STREAMS — довольно устаревшая вещь. Альтернатива сокетам в System V.

Это модель, которая представляет собой поток, у которого есть некая вершина, набор драйверов и конечное устройство.

Базовая единица STREAMS - stream, двунаправленный способ передачи данных между процессом в пользовательском пространстве и драйвером STREAMS в ядре. Stream состоит из заголовка, драйвера и N-го количество модулей.



Попробуем привести пример.

Есть устройство, которое ничего не знает про TCP и UDP, оно знает MAC-адрес получателя и данные. Есть драйвер, который может запаковать IP-адрес в MAC-адрес и запаковать эти данные для сетевого устройства. Драйвер умеет только преобразовывать IP-адрес в MAC-адрес.

Есть драйвер, который реализует работу с TCP и UDP, но ничего не знает про MAC-адреса, поэтому он подготавливает адреса для драйвера, который ниже. И так всё выше и выше, где есть вершина, которой мы говорим, что хотим отправить данные в таком-то направлении.

Соответственно наши данные проходят через ряд настроенных нами драйверов. Эти драйвера оборачивают эти данные заголовками и отправляют в устройства. Устройства этот большой пакет отправляют в сеть.

Модель красивая, потому что одна программа занимается ровно одной задачей. К сожалению, она несостоятельна, потому что постоянно копировать большие объемы данных внутри оперативной памяти даже быстрой всё равно медленно. Поэтому и не используется.

STREAMS могут быть использованы для:

- имплементации сетевых протоколов;
- разработки символьных устройств;
- разработки сетевых контроллеров (например, карты Ethernet);
- ввода-вывода на терминал.

Глава 22

Потоки

Какое есть недостаток у процессов? Для каждого процесса необходимо новое адресное пространство. Порождение процессов — достаточно длительный и затратный процесс с точки зрения операционной системы. Альтернативой процессам являются потоки. Принципиальная разница в том, что они используют общее адресное пространство для нескольких наборов последовательно выполняемых команд.

Поток —

> наименьший набор инструкций, которым может быть выделена квота процессорного времени

Потоки одного процесса имеют единое с ним адресное пространство. Наиболее используемый стандарт —- POSIX. Реализуются семейством функций с префиксом «pthread_»

22.1 Легковесные процессы

Легковесный процесс —

абстракция на уровне ядра ОС для описания процессов, разделяющих единое адресное пространство и системные ресурсы.

22.2 Межпоточное взаимодействие

Для реализации межпоточного взаимодействия можно использовать:

- Любое межпроцессное взаимодействие
- mutex
- rwlock
- volatile переменные
- Общее адресное пространство

Отметим, однако, что не всякий вид межпроцессного взаимодействия разумно использовать для межпоточного взаимодействия.

22.2.1 Общее адресное пространство

22.2.2 Переменные volatile

Для межпоточного взаимодействия можно использовать глобальные переменные, помеченные ключевым словом volatile. Ключевое слово volatile

инструктирует компилятор не оптимизировать доступ к переменной, а каждый раз использовать её значение из памяти.

С одной стороны, это замедляет работу, с другой — гарантируется, что если кто-то другой изменил volatile-переменную, то вы прочитатете ее актуальное значение.

22.2.3 Мьютексы

Мьютекс —

> простейший объект синхронизации, имеющий два состояния: «заблокирован» и «свободен».

При использовании мьютекса только один поток в определенный момент времени может заблокировать мьютекс и получить доступ к разделяемому ресурсу. При завершении работы с ресурсом поток должен разблокировав мьютекс.

Реализованы pthread_mutex.

Приведем некоторые функции для работы с мьютексами:

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *mutex,
          const pthread_mutexattr_t *attr);

int pthread_mutex_destroy(pthread_mutex_t *mutex);

int pthread_mutex_lock(pthread_mutex_t *mutex);

int pthread_mutex_trylock(pthread_mutex_t *mutex);

int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Вам предлагается ознакомиться с ними самостоятельно.

Надо понимать, что используя мьютексы, вы можете эмулировать rwlock . Используя rwlock, вы можете эмулировать семафоры. Используя семафоры — любое поведение. Самое главное — вам нужно определиться, что и для какого случая использовать. Они условно взаимозаменяемы, но всё зависит от того, что конкретно вам нужно от кода.

22.2.4 rwlock

Механизм rwlock — чуть более продвинутая вещь, чем мьютекс. С rwlock вы можете не блокировать ресурс целиком, а, например, сделать его недоступным для записи, но доступным для чтения.

Механизм блокировок rwlock реализуется с помощью таких функций как:

```
#include <pthread.h>

  int pthread_rwlock_init(pthread_rwlock_t * restrict lock,
      const pthread_rwlockattr_t * restrict attr);

  int pthread_rwlock_destroy(pthread_rwlock_t *lock);

int pthread_rwlock_wrlock(pthread_rwlock_t *lock);

  int pthread_rwlock_unlock(pthread_rwlock_t *lock);
```

Литература

1. Ю. Вахалия. UNIX изнутри. – СПб.: Питер, 2003.