Введение в ОС Linux

Это не Windows. Забудьте то, что вы знали раньше

Используемая терминология

- Файловая система представляет собой иерархию файлов и *каталогов*. Не нужно называть каталоги "папками".
- В отличии от Windows, все файлы в UNIX являются равнозначными, независимо от их имени. Понятия "расширение файла" не существует, но для удобства восприятия имени файла пользователем, их снабжают *суффиксами имени*, отделяемые от основного имени точкой. Суффиксов у имени файла может быть несколько, например `.tar.gz`.
- В системе выполняется огромное количество *процессов*, а не "задач". Процессы могут быть запущены как непосредственно пользователем, так и одним из *демонов*, которые запускаются при загрузке системы, и сами по себе являются процессами.

Принятые обозначения клавиатурных сокращений

При работе с консольными UNIX-программами обычно используются следующие, исторически сложившиеся, обозначения клавиатурных сочетаний: * `C-Буква` - одновременное нажатие `Ctrl` и буквенной клавиши. Вниманию пользователей MacOS! `Ctrl` - это именно клавиша `Ctrl`, а не `Command`. * `M-Буква` - одноваременное нажатие `Alt` и буквенной клавиши. Сокращение "M" - от слова "Meta". Такая клавиша была на старых рабочих станциях Sun и SGI. * `C-Буква1 Буква2` - сначала одновременное нажатие `Ctrl` и `Буква1`, затем отпустить клавишу `Ctrl` и нажать `Буква2`. Аналогично для клавиши `Alt`. Сочетание `C-Буква1` называется *префиксом* клавиатурного сочетания, обычно под одинаковыми префиксами группируются клавиатурные сочетания для действий одного характера * `C-Буква1 C-Буква2`. Нажать `Ctrl`, затем нажать и отпустить `Буква1`, нажать и отпустить `Буква2`. После этого можно отпустить клавишу `Ctrl`. * Клавиши `F13` ... `F15`. На PC-клавиатуре их нет. Их нажатие обеспечивается клавишей `Shift` и одной функциональных клавиш с номером `F...` меньше, в зависимости от терминала, на 10 или на 12. Например, во многих графических терминалах, `Shift+F5` означает нажатие клавиши `F15`.

Начало работы

Linux, как и любые другие операционные системы семейства UNIX, является **многопользовательской** операционной системой. Для начала работы необходимо знать свое имя пользователя и пароль.

В зависимости от целей использования, вход в систему может быть осуществлен различными способами.

Локальный вход с графическим интерфейсом

Этот вариант обычно используется при установке Linux в качестве Desktop'а. Как правило, в большинстве дистрибутивов Linux предусмотрен автоматический вход в систему, если при установке был указан только один пользователь-человек (существует еще и другой вид пользователей -- системные). Если пользователей несколько, то вход в систему мало чем отличается от такового в ОС Windows или Mac.

После входа в систему, отображается графическая оболочка (GNOME, Unity или KDE). Командная строка, с которой мы преимущественно будем работать, доступна с помощью приложения "Терминал".

Локальный вход без графического интерфейса

Этот вариант обычно используется при начальной настройке серверов (графический стек является потенциальной "дырой" в безопасности, и обычно не устанавливается), а также при работе со встраиваемыми системами.

После загрузки системы или подкдючения терминала отображается текстовое приглашение с предложением ввести имя пользователя и пароль, а после входа в систему, управление передается **командному интерпретатору**.

Удаленный вход через SSH

Для подключения по SSH необходимо использовать команду (для Linux/Mac)

```
ssh ИМЯ_ПОЛЬЗОВАТЕЛЯ@ИМЯ_ХОСТА
```

Для подключения по SSH из Windows существуют специальные программы, например PuTTY.

После подключения нужно ввести пароль для входа в систему. В некоторых случаях вводить пароль не потребуется, например, если настроена авторизация с использованием SSH-ключей.

После входа в систему, управление передается командному интерпретатору.

Основы работы с командной строкой

Навигация по файловой системе

Приглашение командной строки обычно имеет вид, зависящий от состояния:

```
ИМЯ_ПОЛЬЗОВАТЕЛЯ@ИМЯ_ХОСТА:ТЕКУЩИЙ_КАТАЛОГ>
```

Корневой каталог в иерархии файловой системы -- это `/`. После входа в систему, текущим является *домашний каталог* текущего пользователя, -- это

каталог, который доступен как для чтения, так и для записи.

Домашние каталоги обычных пользователей располагаются в: * `/home/` -- для Linux * `/usr/local/home` -- для FreeBSD * `/Users` -- для MacOS

Независимо от используемой операционной системы, имя `~` (символ "тильда", на одной клавише с буквой "Ё"), является синонимом домашнего каталога *текущего пользователя*.

Лексемы `.` и `..` означают, соответственно, текущий каталог, и каталог на один уровень выше в иерархии.

Для навигации по каталогам используется команда `cd`. Примеры:

```
cd ..          # Перейти на уровень выше
cd ../..       # Перейти на два уровня выше
cd ../src      # Перейти на уровень выше, затем в подкаталог src
cd /           # Перейти в корневой каталог
cd /usr/lib64  # Перейти в каталог /usr/lib64
cd ~/projects  # Перейти в каталог /home/ИМЯРЕК/projects
```

Замечание. При вводе имен файлов или каталогов, клавиша `TAB` вызывает функцию автодополнения имени.

Запуск выполняемых файлов

Выполняемый файл -- это любой файл (в том числе и текстовый), обладающий специальным аттрибутом.

Запуск выполняемого файла осуществляется двумя способами: * Вводом имени этого файла в том случае, если файл располагается в одном из каталогов, перечисленных в *переменной окружения* `PATH`. Все стандартные программы, входящие в поставку UNIX-системы запускаются таким способом. * Вводом *полного имени* файла. Полное имя может быть как абсолютным (то есть начинаться с символа `/`), так и относительным (начинается с символа `.`). Таким способом обычно запускаются программы, находящиеся в домашнем каталоге.

Стандартные программы для управления файлами

- `cp` -- копирование файла или каталога (с опцией `-R`)
- `mv` -- переименование (перемещение) файла или каталога (с опцией `-R`)
- `rm` -- удаление файла или каталога (с опцией `-r`)
- `ls` -- вывод содержимого текущего каталога

Все эти команды являются обычными программами, которые располагаются в каталоге `/usr/bin`.

Вопрос. Почему не существует программы с именем `cd`?

Форматы исполняемых файлов

- Бинарный файл начинается с последовательности байт `0x7F 0x45 0x4C 0x46`. Этот формат называется ELF (Executable and Linkable Format).
- Произвольный файл, в том числе текстовый, который начинается с текстовой строки вида `#!ИМЯ_ИНТЕРПРЕТАТОРА\n`. В этом случае, система запускает указанный интерпретатор, и передает ему выполняемый файл в качестве аргумента.

  Пример выполняемого файла:

  ```
  #!/usr/bin/python

  print("Hello, UNIX!")
  ```

Midnight Commander

Для навигации по файловой системы, использование командной строки, -- это не всегда удобно.

Замечание. При использовании предоставляемого образа ВМ, файловая система также доступна по FTP: ftp://student@192.168.56.105/.

Midnight Commander -- это двухпанельный файловый менеджер, доступный почти для всех UNIX-подобных операционных систем (включая MacOS). Запускается командой `mc` и работа с ним аналогична работе с FAR Manager или Total Commander. Исключение составляют некоторые клавиатурные сочетания.

Основные операции: * `F3` -- просмотр файла * `F4` -- редактирование файла * `Shift+F4` -- создание и редактирование нового файла * `F5` -- копирование * `F6` -- перемещение * `F7` -- создание каталога * `F8` -- удаление * `F10` -- выход из Midnight Commander * `C-x c` -- редактирование аттрибутов файла * `C-x o` -- редактирование пользователя файла * `C-x s` -- создание символической ссылки на файл

Замечание. Выход из редактирования или просмотра файла осуществляется нажатием клавиши `Esc` **два раза**. Это связано с тем, что клавиша `Esc` в классических терминалах предназначена для префиксного ввода управляющих символов.

Иерархия файловой системы

В отличии от Windows, где каждому физическому диску или разделу на диске соответствует определенная буква, например `C:\`, дерево файловой системы в UNIX-системах имеет общий корень `/`. Отдельные диски или разделы *монтируются* в подкаталоги основной файловой системы.

Файловая система всех дистрибутивов Linux имеет следующую иерархию: * `/bin` -- выполняемые программы, предоставляющие минимально

необходимый набор команд * `/boot` -- файлы, необходимые для загрузки операционной системы * `/dev` -- псевдо-файлы устройств * `/etc` -- текстовые файлы настроек * `/home` -- домашние каталоги пользователей * `/lib` или `/lib64`, либо оба этих каталога -- минимально необходимый для работоспособности системы набор разделяемых библиотек. Каталог `/lib64` присутствует на 64-битных системах и содержит варианты библиотек для x86_64, в то время как `/lib` -- их аналоги для i386. * `/lost+found` -- файлы, которые по каким-либо причинам (например, неправильное выключение компьютера, или сбой диска) оказались вне какого-либо каталога, но их содержимое доступно * `/media` -- каталог для монтирования сменных носителей, доступных всем пользователям * `/mnt` -- каталог для монтирования общедоступных сетевых файловых систем или инородных разделов * `/opt` -- каталог для установки сторонних приложений не из репозитория дистрибутива, например Google Chrome или Яндекс.Браузер * `/proc` -- здесь примонтирована виртуальная файловая система с информацией о запущенных в системе процессах * `/root` -- домашний каталог пользователя `root` * `/run` -- содержит *именованные сокеты* и текстовые файлы с *индентификаторами процесса* для запущенных демонов * `/sbin` -- выполняемые файлы для запуска пользователем `root`; у других пользователей этот каталог не включен в переменную окружения `PATH`, и для их запуска необходимо указывать полный путь * `/srv` -- файлы для хранения данных сетевыми службами * `/sys` -- виртуальная файловая система для просмотра и изменения параметров ядра * `/tmp` -- каталог для временных файлов * `/usr` -- содержит иерархию, аналогичную корневой; там находятся файлы большинства программ, которые устанавливаются из *репозиториев* дистрибутива * `/usr/local` -- аналогично `/usr`, но предназначен для установки программ самостоятельно из исходных текстов * `/var` -- содержит данные различных демонов, например базы данных.

Консольные текстовые редакторы

Встроенный редактор Midnight Commander

Вызывается нажатием клавиши `F4` из файлового менеджера, либо командой `mcedit ИМЯ_ФАЙЛА` как самостоятельная программа.

Основные клавиши: * `F2` -- сохранить файл * `Esc Esc` -- выход * `F3` -- начало/конец выделения текста * `F5` -- копирование выделенного текста в текущую позицию * `F6` -- перемещение выделенного текста в текущую позицию * `F8` -- удаление выделенного текста; если текст не выделен, -- удаление текущей строки

Редактор VI

Поскольку Midnight Commander, и соответственно, редактор `mcedit` не всегда установлены по умолчанию, иногда возникает необходимость использования редактора `vi`, который входит в базовый состав почти всех дистрибутивов Linux.

Запускается редактор командой `vi ИМЯ_ФАЙЛА`, либо в результате какого-нибудь действия, которое требует редактирования текста (например, командой `git commit` -- для редактирования комментария к коммиту).

Опознать редактор `vi` можно по черному экрану, в левом столбце терминала при этом во всех пустых строках в конце текста присутствует символ `~`.

После запуска, редактор находится в **командном режиме**. Не нужно нажимать буквенно-цифровые клавиши для ввода текста в этом режиме. Если это все-же произошло, нужно нажать `C-[` для возврата в командный режим.

В командном режиме навигация по тексту осуществляется клавишами стрелок, а также клавишами `h`, `j`, `k`, и `l`. Помимо текстового редактора `vi`, многие среды разработки, например QtCreator и IntelliJ IDEA, а также браузеры имеют Chrome и Firefox имеют плагины, позволяющие использовать навигацию в стиле VIM, так что назначение этих клавиш лучше запомнить.

Для перехода в **режим вставки**, привычному по GUI-редакторам, нужно нажать клавишу `i`. Выход из этого режима -- сочетанием `C-[`. Для перехода в **режим замены** -- клавиша `o`, выход аналогичен.

Основные команды, которые нужно запомнить: * `:w` -- сохранить файл * `:e ИМЯ_ФАЙЛА` -- открыть или создать файл с указанным именем * `:q` -- выход из редактора, возможен только если нет не сохраненных изменений * `:q!` -- принудительный выход из редактора без сохранения * `!КОМАНДА` -- запуск UNIX-команды без выхода из редактора

Более подробное руководство по `vi` можно получить, запустив программу `vimtutor`

Редактор nano

В некоторых дистрибутивах, например Ubuntu, по умолчанию вместо `vi` установлен редактор `nano`. Это простой в использовании текстовый редактор. Опознать его можно по тексту `GNU nano` в заголовке вверху терминала, и подсказкам о сочетаниях клавиш вида `^G Get Help` в подвале. Символ `^` означает клавишу `Ctrl`.

Инструменты разработчика

Компиляторы `gcc` и `clang`

В стандартную поставку современных UNIX-систем входит один из компиляторов: либо `gcc`, либо `clang`. В случае с Linux, по умолчанию обычно используется `gcc`, а в BSD-системах - `clang`. Далее будет описана работа с компилятором `gcc`, имея ввиду, что работа с `clang` ничем принципиально не отличается: у обоих компиляторов много общего, в том числе опции командной строки.

Кроме того, существует команда `cc`, которая является символической ссылкой на используемый по умолчанию компилятор языка Си (`gcc` или `clang`), и команда `c++`, - символическая ссылка на используемый по умолчанию компилятор для C++.

Рассмотрим простейшую программу на языке C++:

```
// файл hello.cpp
#include <iostream>

int
main() {
  std::cout << "Hello, World!" << std::endl;
  return 0;
}
```

Скомпилировать эту программу можно с помощью команды:

```
> c++ -o program.jpg hello.cpp
```

Опция компилятора `-o ИМЯ_ФАЙЛА` указывает имя выходного файла, который нужно создать. По умолчанию используется имя `a.out`. Обратите внимание, что файл `program.jpg` является обычным выполняемым файлом!

Стадии сборки программы на Си или C++

При выполнении команды `c++ -o program.jpg hello.cpp` выполняется достаточно сложная цепочка действий:

1. Выполняется *препроцессинг* текстового файла `hello.cpp`. На этом этапе обрабатываются *директивы препроцессора* (которые начинаются с символа `#`), и получается новый текст программы. Если запустить компилятор с опцией `-E`, то будет выполнен только этот шаг, и на стандартный поток вывода будет выведен преобразованный текст программы.
2. Выполняется *трансляция* одного или нескольких текстов на Си или C++ в объектные модули, содержащие машинный код. Если указать опцию `-c`, то на этом сборка программы будет приостановлена, и будут созданы объектные файлы с суффиксом `.o`. Объектные файлы содержат *бинарный* исполняемый код, которому в точности соотвествует некоторый текст на языке ассемблера. Этот текст можно получить с помощью опции `-S`, - в этом случае, вместо объектных файлов будут созданы текстовые файлы с суффиксом `.s`.
3. Компоновка одного или нескольких объектных файлов в исполняемый файл, и связываение его со стандартной библиотекой Си/C++ (ну и другими библиотеками, если требуется). Для выполнения компоновки компилятор вызывает стороннюю программу `ld`.

Программы на Си v.s. программы на C++

Компилятор `gcc` имеет опцию `-x ЯЗЫК`, для указания языка исходного текста программы: Си (`c`), C++ (`c++`) или Фортран (`fortran`). По умолчанию, язык исходного текста определяется в соответствии с именем файла: `.c`, - это программы на языке Си, а файлы, оканчивающиеся на `.cc`, `.cpp` или `.cxx`, - это тексты на языке C++. Таким образом, имя файла является существенным.

Это относится к стадиям препроцессинга и трансляция, но может вызвать проблемы на стадии компоновки. Например, используя команду `gcc` вместо `g++` (или `cc` вместо `c++`), можно успешно скомпилировать исходный текст программы на C++, но при этом возникнут ошибки на стадии связывания, поскольку компоновщику `ld` будут переданы опции, подразумевающие связывание только со стандартной библиотекой Си, но не C++. Поэтому, при сборке программ на C++ нужно использовать команду `c++` или `g++`.

Указание стандартов

Опция компилятора `-std=ИМЯ` позволяет явным образом указать используемый стандарт языка. Рекомендуется явным образом указывать используемый стандарт, поскольку поведение по умолчанию зависит от используемого дистрибутива Linux. Допустимые имена: * `c89`, `c99`, `c11`, `gnu99`, `gnu11` для языка Си; * `c++03`, `c++11`, `c++14`, `c++17`, `gnu++11`, `gnu++14`, `gnu++17` для языка C++.

Двузначное число в имени стандарта указывает на его год. Если в имени стандарта присутствует `gnu`, то подразумеваются GNU-расширения компилятора, специфичные для UNIX-подобных систем, и кроме того, считается определенным макрос `#define _DEFAULT_SOURCE`, который в некоторых случаях меняет поведение отдельных функций стандартной библиотеки.

В дальнейшем мы будем ориентироваться на стандарт `c11`, а в некоторых задачах, где будет про это явно указано - его расширением `gnu11`.

Объектные файлы, библиотеки и исполняемые файлы

Модуль `ctypes` интерпретатора Python

Рассмотрим программу на языке Си:

```
/* my-first-program.c */
#include <stdio.h>

static void
do_something()
{
    printf("Hello, World!\n");
}

extern void
do_something_else(int value)
{
    printf("Number is %d\n", value);
}

int
main()
{
    do_something();
}
```

Скомпилируем эту программу в объектный файл, а затем - получим из него: (1) выполняемую программу; (2) разделяемую библиотеку. Обратите внимание на опцию `-fPIC`, предназначенную для генерации позиционно-независимого кода, о чем будет рассказано на одном из последующих семинарах.

```
> gcc -c -fPIC my-first-program.c
> gcc -o program.jpg my-first-program.o
> gcc -shared -o library.so my-first-program.o
```

В результате мы получим программу `program.jpg`, которая выводит на экран строку `Hello, World!`, и *библиотеку* с именем `library.so`, которую можно использовать как из Си/C++ программы, так и динамически подгрузить для использования интерпретатором Python:

```
> python3
Python 3.6.5 (default, Mar 31 2018, 19:45:04) [GCC] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from ctypes import cdll
>>> lib = cdll.LoadLibrary("./library.so")
>>> lib.do_something_else(123)
>>> retval = lib.do_something_else(123)
Number is 123
>>> print(retval)
14
```

Обратите внимание, что результатом работы функции `do_something_else` является какое-то загадочное число `14` (возможно, будет какое-то другое при попытке воспроизвести этот эксперимент), хотя функция возвращает `void`.

Причина заключается в том, что разделяемые библиотеки хранят только **имена** функций, но не их сигнатуры (типы параметров и возвращаемого значения).

Попытка вызвать функцию `do_something` не увенчается успехом:

```
>>> lib.do_something()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib64/python3.6/ctypes/__init__.py", line 361, in __getattr__
    func = self.__getitem__(name)
  File "/usr/lib64/python3.6/ctypes/__init__.py", line 366, in __getitem__
    func = self._FuncPtr((name_or_ordinal, self))
AttributeError: ./library.so: undefined symbol: do_something
```

В этом случае имя `do_something` не найдено, поскольку в исходном тексте на языке Си модификатор `static` перед именем функции явно запрещает использование функции где-либо вне текущего исходного текста.

Просмотр таблицы символов

Для исследования объектных файлов, в том числе и скомпонованных, используется утилита `objdump`.

Опция `--syms` или `-t` отображает отдельные секции исполняемого объектного файла, которым присвоены имена - *символы*.

Некоторые имена имеют пометку `*UND*`, - это означает, что имя используется в объектном файле, но его расположение неизвестно. Задача компоновщика состоит как раз в том, чтобы найти требуемые имена в разных объектных файлах или динамических библиотеках, а затем - подставить правильный адрес.

Некоторые символы помечены как глобальные (символ `g` во втором столбце вывода), а некоторые - как локальные (символ `l`). Те символы, которые не являются глобальными, считаются *не экспортируемыми*, то есть (теоретически) не должны быть доступны извне.

Отладчик

Если скомпилировать программу с опцией `-g`, то размер программы увеличится, поскольку в ней появляются дополнительные секции, которые содержат *отладочную информацию*.

Отладочная информация содержит сведения о соответствии отдельных фрагментов программы исходным текстам, и включает в себя номера строк, имена исходных файлов, имена типов, функций и переменных.

Эти информация используется только отладчиком, и почти никак не влияет на поведение программы. Таким образом, отладочную информацию можно совмещать с оптимизацией, в том числе достаточно агрессивной (опция компилятора `-O3`).

Для запуска программы под отладчиком используется команда `gdb`, в качестве аргумента к которой указывается имя выполняемого файла или команды.

Основные команды `gdb`: * `run` - запуск программы, после `run` можно указать аргументы; * `break` - установить точку останова, параметрами этой команды может имя функции или пара `ИМЯ_ФАЙЛА:НОМЕР_СТРОКИ`; * `ni`, `si` - шаг через строку или шаг внутрь функции соответственно; * `return` - выйти из текущей функции наверх; * `continue` - продолжить выполнение до следующей точки останова или исключительной ситуации; * `print` - вывести значение переменной из текущего контекста выполнения.

Взаимодействие с отладчиком производится в режиме командной строки. Различные интегрированные среды разработки (CLion, CodeBlocks, QtCreator) являются всего лишь графической оболочкой, использующей именно этот отладчик, и визуализируя взаимодействие с ним.

Более подробный список команд можно посмотреть в CheatSheet.

Система сборки `cmake`

Использование сторонних библиотек усложняет процесс воспроизводимости сборки. В случае, когда целевая операционная система одна, это не доставляет особых проблем и достаточно простого `Makefile`, но если предполагается разработка кросс-платформенного продукта, то возникают неоднозначности: * какой компилятор используется для сборки (gcc, clang, cl.exe); * расположение include-файлов библиотек для компиляции; * рссположение файлов библиотек для компоновки.

По этой причине часто практикуется не распространение `Makefile`, написанного для конкретного стека инструментов, а его генерация по декларативному описанию в процессе сборки: `./configure`, `qmake`, или `cmake`.

Наиболее гибкой системой сборки, и при этом относительно простой, является CMake, которая реализована с поддержкой не только UNIX-подобных систем, но и Windows.

Проект CMake и его сборка

Описание проекта находится в файле `CMakeLists.txt` и имеет примерно следующий вид:

```
# признак того, что это файл для cmake
# номер версии - это минимально требуемый для сборки проекта
# не стоит злоупотребять указанием самой свежей версии
# CMake, поскольку в консервативных Linux-дистрибутивах
# может быть что-то более старое
cmake_minimum_required(VERSION 3.2)

# имя проекта - не обязательно, обычно используется IDE
project(my_great_project)

# команда `set` устанавливает значение переменной
# некоторые переменные (их имена начинаются с CMAKE_)
# имеют специальное значение

# дополнительные опции компилятора Си
set(CMAKE_C_FLAGS "-std=gnu11")

# дополнительные опции компилятора C++
set(CMAKE_CXX_FLAGS "-std=gnu14")

# в переменной SOURCES будет храниться список файлов;
# если файлов не много, то можно этого не делать,
# но некоторым IDE это требуется для навигации по проекту
set(SOURCES
  file1.c
  file2.cpp
  file3.cpp
)

# добавление цели для сборки - бинарного файла;
# синтаксис ${...} означает использование значения
# переменной, которая в данном примере будет раскрыта
# в список файлов, из которых собирается программа
add_executable(my_cool_program ${SOURCES})
```

Для сборки CMake-проекта необходимо выполнить две стадии: 1. Сгенерировать `Makefile` из `CMakeLists.txt` 2. Собрать проект обычнычным инструментом `make`.

Обычно в процессе генерации `Makefile` и при сборке проекта создается много временных файлов. По этой причине сборку принято проводить в отдельном каталоге, - чтобы не засорять каталог с исходными текстами.

```
$ mkdir build     # создаем каталог для сборки
$ cd build        # переходим в него
$ cmake ../       # генерируем Makefile
                  # аргумент cmake - это каталог, который
                  # содержит файл CMakeLists.txt
$ make            # запуск компиляции
```

Использование сторонних библиотек, для который есть готовое описание CMake

Для многих OpenSource библиотек в стандартной поставке CMake уже готовы модули поддержки, которые выполняют поиск библиотеки. В случае с UNIX этот поиск осуществляется с помощью запуска команд конфигурации, либо проверки различных вариантов написания имен файлов в `/usr/include` и `/usr/lib`. Для Windows просматривается системный реестр.

Список поддерживаемых библиотек можно найти в поставке CMake, для Linux это может быть каталог (в разных дистрибутивах они разные) `/usr/share/cmake/Modules`. Все файлы модулей имеют название `FindИМЯБИБЛИОТЕКИ.cmake`.

Подключение библиотеки, которая поддерживается "из коробки", осуществляется с помощью команды `find_package`. В случае, если необходимые файлы присутствуют, то определяются переменные:

- `ИМЯБИБЛИОТЕКИ_FOUND` - переменная, значение которой устанавливается в `1`, если библиотека не отмечена как `REQUIRED`;
- `ИМЯБИБЛИОТЕКИ_INCLUDE_DIRS` - список дополнительных каталогов, в которых нужно искать заголовочные файлы (опции компилятора `-I...`);
- `ИМЯБИБЛИОТЕКИ_LIBRARIES` - список дополнительных библиотек и каталоги к ним (опции компилятора `-l...` и `-L...`);

Пример для curl:

```
# найти библиотеку CURL; опция REQUIRED означает,
# что библиотека является обязательной для сборки проекта,
# и если необходимые файлы не будут найдены, cmake
# завершит работу с ошибкой
find_package(CURL REQUIRED)

add_executable(my_cool_program ${SOURCES})

# добавляет в список каталогов для цели my_cool_program,
# которые превратятся в опции -I компилятора для всех
# каталоги, которые перечислены в переменной CURL_INCLUDE_DIRECTORIES
target_include_directories(my_cool_program ${CURL_INCLUDE_DIRECTORIES})

# для цели my_cool_program указываем библиотеки, с которыми
# программа будет слинкована
target_link_libraries(my_cool_program ${CURL_LIBRARIES})
```

Если необходимо использовать библиотеку для всех целей проекта, а не для отдельных, то можно использовать команды `include_directories` и `link_libraries`.

Использование сторонних библиотек, для которых есть описания `pkg-config`

Для многих GNU-библиотек существуют описания их использования, которые можно использовать утилитой `pkg-config(1)`. Эти описания можно использовать в проектах CMake в том случае, если для них не реализованы описания CMake, но существуют описания pkg-config. Обычно эти файлы `*.pc` располагаются в каталоге `/usr/lib[64]/pkgconfig`.

Пример для fuse3:

```
# подключаем модуль интеграции с pkg-config
find_package(PkgConfig REQUIRED)

pkg_check_modules(
  FUSE         # имя префикса для названий выходных переменных
  REQUIRED     # если библиотека является обязательной
  fuse3        # имя библиотеки, должен существовать файл fuse3.pc
)

# можно использовать переменные FUSE_INCLUDE_DIRECTORIES
# и FUSE_LIBRARIES
target_include_directories(my_cool_program ${FUSE_INCLUDE_DIRECTORIES})
target_link_libraries(my_cool_program ${FUSE_LIBRARIES})

# дополнительные флаги компиляции, например определения -D...,
# которые не указывают на каталоги для поиска заголовочных файлов,
# перечислены в переменной FUSE_CFLAGS_OTHER
target_compile_options(my_cool_program ${FUSE_CFLAGS_OTHER})
```

Использование сторонних библиотек, для которых вообще ничего нет

В случае, если для библиотеки не подготовлено никаких описаний, то можно попытаться найти необходимые файлы, используя перебор различных комбинаций имен и стандартных каталогов:

```
 # поиск файла динамической библиотеки
find_library(
  SOME_LIBRARY     # переменная, в которую будет записан результат
  NAMES            # перечисляются различные варианты имен для поиска
    some
    something
    somelib0
)

# поиск пути к заголовочным файлам
find_path(
  SOME_INCLUDE_DIRECTORY  # переменная, в которую будет записан результат
  NAMES                   # имена файлов, которые могут содержаться в каталоге
    somelib.h
    somelib_common.h
  PATH_SUFFIXES           # возможные имена подкаталогов в .../include/
    somelib
    somelib-1.0
)

target_include_directories(my_cool_program ${SOME_INCLUDE_DIRECTORY})
target_link_libraries(my_cool_program ${SOME_LIBRARY})
```

Python: расширение и внедрение

Основной источник (на английском): Extending and Embedding.

Справочная информация (на английском): Python/C API.

Ипользование интерпретатора Python

Интерпретатор Python реализован в разделяемой библиотеке, а исполняемый файл интерпретатора `python3` является лишь оболочкой для запуска интерпретатора.

Для сборки программы можно использовать CMake, в стандартной поставке которого входит поддержка Python.

```
find_package(PythonLibs 3.6 REQUIRED)
include_directories(${PYTHON_INCLUDE_DIRS})
target_link_libraries(program ${PYTHON_LIBRARIES})
```

Обратите внимание на то, что необходимо указывать минимальную версию интерпретатора, поскольку в поставку многих дистрибутивах Linux входит две версии Python (2.7 и 3.x), и может возникнуть неоднозначность используемой библиотеки.

Тривиальная реализация своего интерпретатора, с использованием библиотеки `python` выглядит следующим образом:

```
#include <stdio.h>
// В этом заголовочном файле собран почти весь API Python
#include <Python.h>

int main(int argc, char *argv[])
{
    // Открытие файла на чтение
    FILE* fp = fopen(argv[1], "r");
    // Инициализация интерпретатора Python
    Py_Initialize();
    // Выполнение файла
    PyRun_SimpleFile(fp, argv[1]);
    // Завершение работы интерпретатора
    Py_Finalize();
    // Закрытие файла
    fclose(fp);
}
```

Указание имени файла в качестве второго аргумента является желательным по двум причинам: оно используется при генерации сообщении возникающих исключительных ситуаций, и кроме того, используется для определения пути поиска зависимых модулей. Переданный текст доступен через глобальную переменную `__file__` и может быть произвольным.

```
/* Си */
PyRun_SimpleFile(fp, "abrakadabra");

# Python
print(__file__)
# abrakadabra
```

Скрипт на языке Python может иметь аргументы командной строки, которые доступны через переменную-список `sys.argv`. В приведенной выше тривиальной реализации интерпретатора это значение не установлено, поэтому обращение к `sys.argv` выдаст ошибку о том, что эта переменная не определена:

```
AttributeError: module 'sys' has no attribute 'argv'
```

Перед запуском файла на выполнение можно установить список аргументов с помощью `PySys_SetArgv`:

```
wchar_t* args[] = { L"One", L"Two", L"Аргумент" };
PySys_SetArgv(3, args); // int argc, wchar_t *argv[]
```

Обратите внимание на то, что строки в Python являются многобайтовыми, поэтому многие функции API подразумевают работу с типом данных `wchar_t`. В случае использования однобайтных цепочек символов используется системная локаль, как правило это UTF-8.

Программой может быть не только текст программы, хранящийся в файле, но и произвольная строка текста:

```
PyRun_SimpleString("a=1\nb=2\nprint(a+b)");
// будет выведено 3
```

При выполнении текста, с которым не связан никакой файл, глобальная переменная `__file__` считается не определенной, а в случае возникновения исключений, в качестве файла-источника будет использован текст `<string>`.

```
PyRun_SimpleString("print(__file__)");

Traceback (most recent call last):
  File "<string>", line 1, in <module>
NameError: name '__file__' is not defined
```

API стандартных классов Python

Язык Python содержит реализацию стандартных контейнерных классов, которые являются встроенными типами Python, но их можно использовать и без интерпретации текста программы.

Базовым классом для всех объектов является класс `object`, которому в Си API соответствует базовый класс `PyObject`. Поскольку интерптетатор реализован на языке Си, который не является объектно-ориентированным, то на пользователя API возлагается ответственноть за контролем используемых типов.

Все классы и методы в PyObject API именуются следующим образом: `PyКласс_Метод`, а указатель на объект класса передается в качестве первого аргумента.

Примеры стандартных классов и методов:

- `PyList_Append(PyObject *list, PyObject *item)` - эквивалент `list.append(item)`
- `PyDict_SetItem(PyObject *p, PyObject *key, PyObject *val)` - эквивалент `p[key]=val`

Обратите внимание, что для объектов всех классов используется тип `PyObject*`, поэтому необходимо предварительно проверять, какой тип имеет переменная с помощью одной из функций вида `PyКласс_Check(PyObject *p)`, которая возвращает ненулевое значение в случае принадлежности объекта к классу, и `0` в противном случае.

Соответствие стандартных типов Python префиксам функций PyObject API: `list` - `PyList_`, `tuple` - `PyTuple_`, `dict` - `PyDict_`, `str` - `PyUnicode_`, `bytes` - `PyBytes_`, `file` - `PyFile_`, `int` - `PyLong_`, `float` - `PyFloat_`.

Обратите внимание, что тип для строк называется `PyUnicode`, а не `PyString`. Это связано с тем, что во времена Python 2 строки были двух видов: однобайтные и юникодные, а в Python 3 остались только юникод-строки.

Пример использования API без интерпретатора: разбить текст на лексемы, выделяя целые числа как числа, а остальные слова оставляя строками.

Этому коду соответствует программа на Python:

```
text = "сейчас 23 59 не время спать"
result = []
tokens = text.split(" ")
for entry in tokens:
    try:
        number = int(entry)
        result += [number]
    except:
        result += [ entry.upper() ]
print(result)
# ['сейчас', 23, 59, 'не', 'время', 'спать']
```

```
int main()
{
    // Если не используются wchar_t*, то по умолчанию подразумевается,
    // что все однобайтные строки - в кодировке UTF-8
    static const char TheText[] = "сейчас 23 59 не время спать";

    // Инициализация API Python
    Py_Initialize();

    // Создание Python-строк из Си-строк
    PyObject *py_text = PyUnicode_FromString(TheText);
    PyObject *py_space_symbol = PyUnicode_FromString(" ");

    // Создание пустого списка
    PyObject *py_result = PyList_New(0);
    // str.split(py_text, py_space_symbol, maxsplit=-1)
    PyObject *py_tokens = PyUnicode_Split(py_text, py_space_symbol, -1);
    PyObject *py_entry = NULL;
    PyObject *py_number = NULL;

    // Цикл по элементам списка. PyList_Size - его размер
    for (int i=0; i<PyList_Size(py_tokens); ++i) {
        // list.__getitem__(i) - этому методу соответствует оператор []
        py_entry = PyList_GetItem(py_tokens, i);
        // Попытка создать int из строки, base=10
        // В случае не успеха устанавливается ошибка ValueError
        py_number = PyLong_FromUnicodeObject(py_entry, 10);
        // Проверяем, не возникло ли исключение
        if (! PyErr_Occurred()) {
            // OK - преобразование int(py_entry) выполнено успешно
            PyList_Append(py_result, py_number);
        }
        else {
            // Возникло исключение, оставляем просто текст
            PyList_Append(py_result, py_entry);
            // Убираем флаг ошибки, так как мы её обработали.
            // Если этого не сделать, то это исключение попадет
            // в интерпретатор, как только он будет использован
            PyErr_Clear();
        }
    }
    // Вывод print(repr(py_result))
    // Если последний параметр Py_PRINT_RAW вместо 0,
    // то вместо repr() будет использована функция str() для
    // преобразования произвольного объекта к строковому виду
    PyObject_Print(py_result, stdout, 0);

    Py_Finalize();
}
```

Расширениие функциональности Си-модулями

Скрипты на Python, выполняемые интерпретатором, могут использовать любые модули через `import`, в этом случае выполняется их поиск в одном из каталогов, перечисленных в списке `sys.path`. Некоторые из модулей являются *встроенными* (built-in), и не загружаются из внешних файлов, а создаются самим интерпретатором.

Для доступа к функциональности приложения, в который встраивается интерпретатор, можно использовать встроенные модули, которые взаимодействуют с самим приложением.

```
# создадим модуль с названием 'app', который реализует функциональность
import app
# модуль может содержать функции
app.do_something()
# и какие-нибудь глобальные переменные
print(app.some_value)
```

При выполнении строки `import` интерпретатор выполняет поиск модуля, и в случае успеха, создает новый объект модуля с указанным именем, используя его в качестве глобального пространства имен, в котором выполняется инициализация модуля.

Иниализация выполняется ровно один раз, независимо от того, сколько раз импортируется модуль. Для обычных Python-модулей, его инициализация заключается в выполнении текста программы, а для встроенных модулей - в вызове функции, которая возвращает новый модуль.

```
static PyObject *
create_module() {
    // NULL в качестве возвразаемого значения любой функции,
    // которая должна возвращать PyObject*, означает
    // исключительную ситуацию
    PyErr_SetString(PyExc_RuntimeError, "Not implemented yet");
    return NULL;
}

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

    // Добавляем в таблицу имя встроенного модуля
    PyImport_AppendInittab("app", create_module);

    // Регистрация встроенных модулей должна быть сделана
    // раньше, чем PyInitialize
    PyInitialize();
    ...
}
```

Сам модуль - это объект Python, который инициализируется из структуры-описания `PyModuleDef`:

```
static PyModuleDef moduleDef = {
    // ссылка на RTTI, поскольку Си не является ООП-языком
    .m_base = PyModuleDef_HEAD_INIT,
    // имя модуля
    .m_name = "app",
    // размер дополнительной памяти для хранения состояния модуля в
    // случае использования нескольких интерпретаторов, либо -1,
    // если не планируется использование PyModule_GetState
    .m_size = -1,
    // указатель на список методов (функций) модуля, может быть NULL
    .m_methods = methods,
};
PyObject *module = PyModule_Create(&moduleDef);
```

Список методов модуля - это массив объектов `PyMethodDef`, признаком конца которого является "нулевой элемент", - структура заполненная нулями, по аналогии с признаком конца строк в языке Си.

```
static PyObject *
do_something(PyObject *self, PyObject *args) {
    PyErr_SetString(PyExc_RuntimeError, "Not implemented yet");
    return NULL;
}

static PyMethodDef methods[] = {
    {
        // имя Python-функции
        .ml_name = "do_something",
        // указатель на Си-функцию
        .ml_meth = do_something,
        // флаги использования Си-функции
        .ml_flags = METH_VARARGS,
        // строка описания, выдается функцией help()
        .ml_doc = "Do something very useful"
    },
    // признак конца массива описаний методов
    {NULL, NULL, 0, NULL}
};
```

Каждая Си-функция, которая реализует Python-функцию, должна возвращать объект `PyObject*`, и принимает минимум один аргумент - указатель на сам объект модуля.

Си-функции могут иметь разные аргументы (включая из количество), в зависимости о того, как допускается вызывать метод. Это поведение определяется флагами в поле `ml_flags`.

- С одним аргументом: `(PyObject *self)` - в случае, если функция не принимает никаких аргументов и значение в `.ml_flags = METH_NOARGS`
- С двумя аргументами: `(PyObject *self, PyObject *argsTuple)`, причем второй аргумент является кортежем (возможно пустым), - в случае если функция принимает переменное количество позиционных аргументов и значение в `.ml_flags = METH_VARARGS`
- С двумя аргументами: `(PyObject *self, PyObject *argsDict)`, причем второй аргумент является словарем (возможно пустым), - в случае если функция принимает переменное количество именованных аргументов и значение в `.ml_flags = METH_KEYWORDS`
- С тремя аргументами: `(PyObject *self, PyObject *argsTuple, PyObject *argsDict)`, где второй аргумент - это кортеж из позиционных аргументов, а третий - это словарь именованных аргументов, - при значении `.ml_flags = METH_KEYWORDS|METH_VARARGS`

Возвращаемое значение `NULL` вместо объекта `PyObject*` означает исключительную ситуацию. В языке Python любая функция должна возвращать хоть какой-нибудь объект. С точки зрения синтаксиса языка, отсутствие возвращаемого значения означает, что будет возвращен объект типа `NoneType`, который называется `None`.

Объект типа `None` существует в единственном экзепляре на весь интерпретатор, но при этом может много где ипользоваться, и к нему, как и к любому объекту, применяются обычные правила подсчета ссылок.

```
def a(): pass  # функция, которая "ничего не возвращает"
b = a()        # b = None, причем вызов a() увеличил счетчик ссылок на None

a()            # возращается None, и результат отбрасывается,
               # поскольку он ничему не присвоен. В момент вызова a()
               # увеличивается счетчик ссылок, при отсутствии левой части
               # присваивания счетчик сслок уменьшается
```

При создании новых объектов из Си-кода, счетчик ссылок устанавливается равным 1, и обычно никаких действий по его увеличению не требуется, если объекты возвращаются интерпретатору:

```
static PyObject *
func_returning_string(PyObject *self)
{
    PyObject *ret = PyUnicode_FromString("Hello"); // ret->ob_refcnt=1
    return ret; // OK
}

# из Python:
a = func_returning_string()  # ret -> a, refcnt=1
func_returning_string()      # del ret, refcnt=1 --> refcnt=0
```

В случае использования `None`, поскольку он существует в единственном экземпляре, нужно увеличить количество ссылок:

```
static PyObject *
func_returning_none(PyObject *self)
{
    // Py_None - это указатель на статический объект _Py_NoneStruct
    Py_INCREF(Py_None);   // Py_None->ob_refcnt ++
    return Py_None;
}
```

Реализация модулей для использования штатным интерпретатором

Си-модуль для языка Python - это разделяемая библиотека, которая загружается через механизм `dlopen`, и поэтому должна быть скомпилирована в позиционно-независимый код (опция `-fPIC` компилятора).

Библиотека имеет не стандартное имя файла:

- `МОДУЛЬ.so` - для Mac, Linux и *BSD. Обратите внимание на отсутствие префикса `lib` в имени, и кроме того, в Mac используется суффикс `.so` вместо `.dynlib`
- `МОДУЛЬ.pyd` или `МОДУЛЬd.pyd` - для Windows. Вместо суффикса `.dll` используется `.pyd` или `d.pyd` (для варианта сборки с отладочной информацией).

Единственная функция, которая обязана быть реализована в библиотеке - это функция:

```
PyObject* PyInit_МОДУЛЬ();
```

Функция должна создавать и возвращать объект модуля, по аналогии с расширением интерпретатора встроенным модулем.

Если код модуля реализуется на языке C++, то необходимо отключить преобразование имен с помощью `extern "C"`, а в случае с операционной системой Windows и компилятором MSVC - ещё и объявить функцию экспортируемой: `__declspec(dllexport)`.

Все платформо-зависимые объявления спрятаны в макрос `PyMODINIT_FUNC`, значение которого определяется препроцессором. Таким образом, модуль реализуется функцией:

```
PyMODINIT_FUNC PyInit_my_great_module() {
    static PyModuleDef modDef = {
        .m_base = PyModuleDef_HEAD_INIT,
        .m_name = "my_great_module",
        ....
    }
    return PyModule_Create(&modDef);
}
```

Обратите внимание, что имя файла с модулем, имя части функции после `PyInit_` и имя самого модуля должны совпадать, иначе интерпретатор не сможет найти и загрузить его.

Все остальные функции модуля могут быть статическими, а не экспортироваться из библиотеки, поскольку указатели на них явным образом будут присутствовать в объекте, который вернет функция инициализации.

В CMake-пакете `PythonLibs` определяется функция для создания цели-модуля:

```
find_package(PythonLibs 3.6 REQUIRED)

python_add_module(my_great_module module.c)
```

Эта цель определяет необходимые опции компиляции для сборки позиционно-независимого кода с правильным именем, независимо от используемой операционной системы.

Для загрузки модуля из Python необходимо разместить его в одном из каталогов поиска модулей Python, либо рядом с файлом скрипта, который его использует. Python при этом корректно работает с символическими ссылками.

Python и отладчик GDB

Отладчик GDB позволяет ставить точки останова в любой части программы, для которой существует отладочная информация. Таким образом, если собрать модуль отдельно с опцией `-g`, то вместе с ним можно использовать `gdb` даже в том случае, если для самого интерпретатора отладочная информация отсутствует. Целевой программой для `gdb` указывается исполняемый файл интерпретатора `python3`, а сам тестовый скрипт - в качестве аргумента запуска.

```
> gdb python3
(gdb) b module.c:112
(gdb) r script.py
```

Современные версии `gdb` (начиная с 7.x) включают поддержку расширений для типов данных Python API, и использование команды отладчика `print` вызывает метод `repr` языка Python для выводимых объектов, а он, в свою очередь - подразумевает вызов функции `PyObject_Repr(PyObject *obj)` для произвольного Python-объекта.

В случае, если переменная не инициализирована, и содержит мусор по указателю, то это может приводить к ошибке нарушения сегментации, причиной которой становится сам отладчик, вызывая `print` . При использовании интегрированных сред разработки, эта команда отладчика вызывается очень часто для обновления значений локальных переменных, что может приводить к печальным последствиям.

```
PyObject* some_function(PyObject *self, PyObject *args) {
    // <-- точка останова где-то здесь
    ....
    ....
    PyObject * value = ...
    ...
}
```

В данном примере отладчик инициирует ошибку нарушения сегментации, поскольку локальная переменная `value` ещё не инициализирована. Для того, чтобы этого избежать, есть два способа:

- Отключить использования вызова `repr` для Python-объектов командой отладчика `disable pretty-printer` (в среде QtCreator это делается автоматически при снятии чекбокса "Use Debugging Helper" в настройках отладчика), - в этом случае все Python-объекты будут отображаться как Си-структуры.
- Реорганизовать код таким образом, чтобы на момент остановки отладчиком все локальные переменные были инициализированы.

  ```
  PyObject* some_function(PyObject *self, PyObject *args) {
    PyObject * value = NULL;
    ....
    // <-- точка останова после инициализации всех PyObject*
    ....
    value = ...
    ...
  }
  ```

Целочисленная арифметика

Целочисленные типы данных

Минимально адресуемым размером данных является, какправило, один байт (8 бит). Как правило - это значит, что не всегда, и бывают разные экзотические архитектуры, где "байт" - это 9 бит (PDP-10), или специализированные сигнальные процессоры с минимально адресуемым размером данных 16 бит (TMS32F28xx).

По стандарту языка Си определена константа `CHAR_BIT` (в заголовочном файле `<limits.h>`), для которой гарантируется, что `CHAR_BIT >= 8`.

Тип данных, представляющий один байт, исторически называется "символ" - `char`, который содержит ровно `CHAR_BIT` количество бит.

Знаковость типа `char` по стандарту не определена. Для архитектуры x86 это знаковый тип данных, а, например, для ARM - беззнаковый. Опции компилятора gcc `-fsigned-char` и `-funsigned-char` определяют это поведение.

Для остальных целочисленных типов данных: `short`, `int`, `long`, `long long`, стандарт языка Си определяет минимальную разрядность:

Тип данных	Разрядность
`short`	не менее 16 бит
`int`	не менее 16 бит, обычно 32 бит
`long`	не менее 32 бит
`long long`	не менее 64 бит, обычно 64 бит

Таким образом, полагаться на количество разрядов в базовых типах данных нельзя, и это нужно проверять с помощью оператора `sizeof`, который возвращает "количество байт", то есть, в большинстве случает - сколько блоков размером `CHAR_BIT` помещается в типе данных.

С особой осторожностью нужно относиться к типу данных `long`: на 64-разрядной системе Unix он является 64-битным, а, например, на 64-битной Windows - 32-битным. Поэтому, во избежание путаницы, использовать этот тип данных запрещено.

Знаковые и беззнаковые типы данных

Перед целочисленными типами данных могут стоять модификаторы `unsigned` или `signed`, которые указывают допустимость отрицательных чисел.

Для знаковых типов, старший бит определяет знак числа: значение `1` является признаком отрицательности.

Способ внутреннего представления отрицательных чисел не регламентирован стандартом языка Си, однако все современные компьютеры используют обратный дополнительный код. Более того, п.6.3.1.3.2 стандарта языка Си определяет способ приведения типов от знакового к беззнаковому таким способом, которые приводит к кодированию обратным дополнительным кодом.

Таким образом, значение `-1` представляется как целое число, все биты которого равны единице.

С точки зрения низкоуровневого программирования, и языка Си в частности, знаковость типов данных определяет только способ применения различных операций.

Типы данных с фиксированным количеством бит

В заголовочных файлах файле `<stdint.h>` (для Си99+) и `<cstdint>` (для C++11 и новее) определены типы данных, для которых гарантируется фиксированное количесвто разрядов: `int8_t`, `int16_t`, `int32_t`, `int64_t`, - для знаковых, и `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t` - для беззнаковых.

Переполнение

Ситуация целочисленного переполнения возникает, когда тип данных результата не имеет достаточно разрядов для того, чтобы хранить итоговый результат. Например, при сложении беззнаковых 8-разрядных целых чисел 255 и 1, получается результат, который не может быть представим 8-разрядным значением.

Для **беззнаковых чисел** ситуация переполнения является штатной, и эквивалентна операции "сложение по модулю".

Для **знаковых** типов данных - приводит к ситуации *неопределенного поведения* (Undefined Behaviour). В корректных программах такие ситуации встречаться не могут.

Пример:

```
int some_func(int x) {
    return x+1 > x;
}
```

С точки зрения здравого смысла, такая программа должна всегда возвращать значение `1` (или `true`), поскольку мы знаем, что `x+1` всегда больше, чем `x`. Компилятор может использовать этот факт для оптимизации кода, и всегда возвращать истинное значение. Таким образом, поведение программы зависит от того, какие опции оптимизации были использованы.

Контроль неопределенного поведения

Свежие версии компиляторов `clang` и `gcc` (начиная с 6-й версии) умеют контролировать ситуации неопределенного поведения.

Можно включить генерацию *управляемого* кода программы, который использует дополнительные проверки во время выполнения. Естественно, это происходит ценой некоторого снижения производительности.

Такие инструменты называются *ревизорами* (sanitizers), предназначенными для разных целей.

Для включения ревизора, контролирующего ситуацию неопределенного поведения, используется опция `-fsanitize=undefined`.

Контроль переполнения, независимо от знаковости

Целочисленное переполнение означает перенос старшего разряда, и многие процессоры, включая семейство x86, позволяют это диагностировать. Стандартами языков Си и C++ эта возможность не предусмотрена, однако компилятор gcc (начиная с 5-й версии) предоставляет **нестандартные** встроенные функции для выполнения операций с контролем переполнения.

```
// Операция сложения
bool __builtin_sadd_overflow (int a, int b, int *res);
bool __builtin_saddll_overflow (long long int a, long long int b, long long int *res);
bool __builtin_uadd_overflow (unsigned int a, unsigned int b, unsigned int *res);
bool __builtin_uaddl_overflow (unsigned long int a, unsigned long int b, unsigned long int *res);
bool __builtin_uaddll_overflow (unsigned long long int a, unsigned long long int b, unsigned long long int *res);

// Операция вычитания
bool __builtin_ssub_overflow (int a, int b, int *res)
bool __builtin_ssubl_overflow (long int a, long int b, long int *res)
bool __builtin_ssubll_overflow (long long int a, long long int b, long long int *res)
bool __builtin_usub_overflow (unsigned int a, unsigned int b, unsigned int *res)
bool __builtin_usubl_overflow (unsigned long int a, unsigned long int b, unsigned long int *res)
bool __builtin_usubll_overflow (unsigned long long int a, unsigned long long int b, unsigned long long int *res)

// Операция умножения
bool __builtin_smul_overflow (int a, int b, int *res)
bool __builtin_smull_overflow (long int a, long int b, long int *res)
bool __builtin_smulll_overflow (long long int a, long long int b, long long int *res)
bool __builtin_umul_overflow (unsigned int a, unsigned int b, unsigned int *res)
bool __builtin_umull_overflow (unsigned long int a, unsigned long int b, unsigned long int *res)
bool __builtin_umulll_overflow (unsigned long long int a, unsigned long long int b, unsigned long long int *res)
```

Представление вещественных чисел

Существует два способа представления вещественных чисел: с фиксированным количеством разрядов (fixed-point) под дробную часть, и с переменным числом разрядов (floating-point).

Представление чисел с фиксированной точкой часто используется там, где требуется гарантированная точность до определенного разряда, например, в финансовой сфере.

Представление в формате с плавающей точкой является более универсальным, и все современные архитектуры процессоров работают именно в этом формате.

Числа с плавающей точкой в формате IEE754

Два основных типа вещественных с плавающей точкой, которые определены стандартом языка Си, - это `float` (используется 4 байта для хранения) и `double` (используется 8 байт).

Самый старший бит в представлении числа - это признак отрицательного значения. Далее, по старшинству бит, хранится значения *смещенной экспоненциальной* части (8 бит для `float` или 11 бит для `double`), а затем - значение *мантиссы* (23 или 52 бит).

Смещение экспоненциальной части необходимо для того, чтобы можно было в таком представлении хранить значения с отрицательной экспонентой. Смещение для типа `float` равно `127`, для типа `double` - `1023`.

Таким образом, итоговое значение может быть получено как:

```
Value = (-1)^S * 2^(E-B) * ( 1 + M / (2^M_bits - 1) )
```

где `S` - бит знака, `E` - значение смещенной экспоненты, `B` - смещение (127 или 1023), а `M` - значение мантиссы, `M_bits` - количество бит в экспоненте.

Как получить отдельные биты вещественного числа

Поразрядные операции относятся к целочисленной арифметике, и не предусмотрены для типов `float` и `double`. Таким образом, нужно сохранить вещественное число в памяти, и затем прочитать его, интерпретируя как целое число. В случае с языком C++ для этого предназначен оператор `reinterpret_cast`. Для языка Си есть два способа: использовать аналог `reinterpret_cast` - приведение указателей, либо использовать тип `union`.

Приведение указателей

```
// У нас есть некоторое целое вещественное число, которое хранится в памяти
double a = 3.14159;

// Получаем указатель на это число
double* a_ptr_as_double = &a;

// Теряем информацию о типе, приведением его к типу void*
void* a_ptr_as_void = a_ptr_as_void;

// Указатель void* в языке Си можно присваивать любому указателю
uint64_t* a_ptr_as_uint = a_ptr_as_void;

// Ну а дальше просто разыменовываем указатель
uint64_t b = *a_as_uint;
```

Использование типа `union`

Тип `union` - это тип данных, который синтаксически очень похож на тип `struct`, то есть там можно перечислить несколько именованных полей, но концептуально - это совершенно разные типы данных! Если в структуре или классе, для хранения каждого поля для предусмотрено отдельное место в памяти, то для `union` этого не происходит, и все поля накладываются друг на друга при размещении в памяти.

Обычно тип `union` используется в качестве вариантного типа данных (в C++ начиная с 17-го стандарта для этого предусмотрен `std::variant`), но в качестве побочного эффекта - его удобно использовать приведения типов в стиле `reinterpret_cast`, не используя при этом указатели.

```
// У нас есть некоторое целое вещественное число, которое хранится в памяти
double a = 3.14159;

// Используем тип union
typedef union {
    double     real_value;
    uint64_t   uint_value;
} real_or_uint;

real_or_uint u;
u.real_value = a;
uint64_t b = u.uint_value;
```

Специальные значения в формате IEEE754

- Бесконечность: `E=0xFF...FF`, `M=0`
- Минус ноль (результат деления 1 на минус бесконечность): `S=1`, `E=0`, `M=0`
- NaN (Not-a-Number): `S` - любое, `E=0xFF...FF`, `M <> 0`

Некоторые процессоры, например архитектуры x86, поддерживают расширение стандарта, позволяющее более эффективно представлять множество чисел, значения которых близко к нулю. Такие числа называются *денормализованными*.

Признаком денормализованного числа является значение смещенной экспоненты `E=0`. В этом случае, численное значение получается следующим образом:

```
Value = (-1)^S * ( M / (2^M_bits - 1) )
```

Значения Not-a-Number

Некоторые процессоры, например Intel x86, различают два вида чисел `NaN` - невалидное значение.

sNaN - Signaling NaN

Значения `sNaN` возникают при выполнении операций, которые сигнализируют об ошибке на уровне прерывания процессора. Например, деление на `0`. Обычно, чтобы получить такие значения, необходимо собирать программу с опцией `-fno-signaling-nans`. Более подробно - см. FloatingPointMath - GCC Wiki

Пример битовой маски для типа `double` на x86_64, определяющей значение `sNaN`:

```
Номера битов    6         5         4         3         2         1         0
             3210987654321098765432109876543210987654321098765432109876543210
             ----------------------------------------------------------------
Значения     0111111111110100000000000000000000000000000000000000000000000000
             ----------------------------------------------------------------
Регион       SEEEEEEEEEEEMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
```

qNaN - Quiet NaN

В отличии от `sNaN`, значения Quiet NaN, которые получаются в результате вычислений, не приводят к прерыванию процессора и вызове обработчика исплючительной ситуации.

Примером является попытка сложить `+inf` и `-inf`.

Пример битовой маски для типа `double` на x86_64, определяющей значение `qNaN`:

```
Номера битов    6         5         4         3         2         1         0
             3210987654321098765432109876543210987654321098765432109876543210
             ----------------------------------------------------------------
Значения     0111111111111000000000000000000000000000000000000000000000000000
             ----------------------------------------------------------------
Регион       SEEEEEEEEEEEMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
```

Разработка под архитектуру ARM

Кросс-компиляция

Процесс сборки программ, предназначенных для другой процессорной архитектуры или операционной системы называется кросс-компиляцией.

Для этого необходимо специальная версия компилятора `gcc`, предназначенного для другой платформы. Во многих дистрибутивах существуют отдельные пакеты компилятора для других платформ, включая ARM.

Кроме того, для архитектуры ARM можно скачать готовую поставку "все-в-одном" из проекта Linaro: http://releases.linaro.org/components/toolchain/binaries/7.3-2018.05/arm-linux-gnueabi/.

Полные названия команд `gcc` имеют вид *триплетов*:

```
ARCH-OS[-VENDOR]-gcc
ARCH-OS[-VENDOR]-g++
ARCH-OS[-VENDOR]-gdb

и т. д.
```

где `ARCH` - это имя архитектуры: `i686`, `x86_64`, `arm`, `ppc` и т.д.; `OS` - целевая операционная система, например `linux`, `win32` или `darwin`; а необязательный фрагмент триплета `VENDOR` - соглашения по бинарному интерфейсу, если их для платформы существует несколько, например для ARM это может быть `gnueabi` (стандартное соглашение Linux) или `none-eabi` (без операционной системы, просто голое железо).

Для ARM ещё часто различают название архитектуры на `arm` (soft float) и `armhf` (hard float). В первом случае подразумевается отсутствие блока с плавающей точкой, поэтому все операции эмулируются программно, во втором случае - выполняются аппаратно.

Выполнение программ для не родных архитектур

Выполнение программ, предназначенных для других архитектур, возможно только интерпретацией инородного набора команд. Для этого предназначены специальные программы - *эмуляторы*.

Архитектуру ARM, как и многие другие архитектуры, поддерживает эмулятор QEMU.

Эмулировать можно как компьютерную систему целиком, по аналогии с VirtualBox, так и только набор команд процессора, используя при этом окружение хост-системы Linux.

Запуск бинарников ARM в родном окружении

Этот эмулятор входит в состав всех распространенных дистрибутивов. Команды qemu имеют вид:

```
qemu-ARCH
qemu-system-ARCH
```

где `ARCH` - это имя эмулируемой архитектуры. Команды, в названии которых присутствует `system`, запускают эмуляцию компьютерной системы, и для их использования необходимо установить операционную систему.

Команды без `system` требуют в качестве обязательного аргумента имя выполняемого файла для ОС Linux, и эмулируют только набор команд процессора в *пользовательском режиме*, выполняя "инородный" исполняемых файл так, как будто это обычная программа.

Поскольку большинство программ, скомпилированных для ARM Linux, подразумевают использование стандартной библиотеки Си, необходимо использовать именно версию glibc для ARM. Минимальное окружение с необходимыми библиотеками можно взять из проекта Linaro (см. ссылку выше), и скормить его qemu с помощью опции `-L ПУТЬ_К_SYSROOT`.

Пример компиляции и запуска:

```
# в предположении, что компилятор распакован в /opt/arm-gcc,
# а sysroot - в /opt/arm-sysroot

# Компилируем
> /opt/arm-gcc/bin/arm-linux-gnueabi-gcc -marm -o program hello.c

# На выходе получаем исполняемый файл, который не запустится
> ./program
bash: ./program: cannot execute binary file: Exec format error

# Но мы можем запустить его с помощью qemu-arm
> qemu-arm -L /opt/arm-sysroot ./program
Hello, World!
```

Запуск ARM-программ в эмуляции окружения Raspberry Pi

Идеальный вариант для тестирования и отладки - это использовать настоящее железо, например Raspberry Pi.

Если под рукой нет компьютера с ARM-процессором, то можно выполнять эмуляцию ПК с установленной системой Raspbian.

Скачать образ можно отсюда: гуглодиск

Основы ассемблера ARM

Написание и компиляция программ

Программы на языка ассемблера для компилятора GNU сохраняются в файле, имя которого оканчивается на `.s` или `.S`. Во втором случае (с заглавной буквой) подразумевается, что текст программы может быть обработан препроцессором.

Для компиляции используется одна из команд: `arm-linux-gnueabi-as` или `arm-linux-gnueabi-gcc`. В первом случае текст только компилируется в объектный файл, во втором - в выполняемую программу, скомпонованную со стандартной библиотекой Си, из которой можно использовать функции ввода-вывода.

Процессоры ARM поддерживают два набора команд: основной 32-битный `arm`, и уплотнённый 16-битный `thumb`, между которыми процессор умеет переключаться. В рамках данного семинара мы будем использовать 32-битный набор инструкций, поэтму тексты нужно компилировать с опцией `-marm`.

Общий синтакис

```
// Это комментарий, как в C++

    .text      // начало секции .text с кодом программы
    .global f  // указание о том, что метка f
               // является доступной извне (аналог extern)

f:             // метка (заканчивается двоеточием)

     // последовательность команд
     mul   r0, r0, r3
     mul   r0, r0, r3
     mul   r1, r1, r3
     add   r0, r0, r1
     add   r0, r0, r2
     mov   r1, r0
     bx    lr
```

Регистры

Процессор может выполнять операции только над *регистрами* - 32-биьными ячейками памяти в ядре процессора. У ARM есть 16 регистров, доступных программно: `r0`, `r1`, ... , `r15`.

У регистров `r13` ... `r15` имеются специальные назначения и дополнительные имена:

- `r15` = `pc` : Program Counter - указатель на текущую выполняемую инструкцию
- `r14` = `lr` : Link Register - хранит адрес возврата из функции
- `r13` = `sp` : Stack Pointer - указатель на вершину стека.

Флаги

Выполнение команд может приводить к появлению некоторой дополнительной информации, которая хранится в *регистре флагов*. Флаги относятся к последней выполненной команде. Основные флаги, это:

- `C` : Carry - возникло беззнаковое переполнение
- `V` : oVerflow - возникло знаковое переполнение
- `N` : Negative - отрицательный результат
- `Z` : Zero - обнуление результата.

Команды

Полный перечень 32-битных команд см. в этом reference, начиная со 151 страницы.

Архитектура ARM-32 подразумевает, что почти команды могут иметь *условное выполнение*. Условие кодируется 4-мя битами в самой команде, а с точки зрения синтаксиса ассемблера у команд могут быть суффиксы.

Таким образом, каждая команда состоит из двух частей (без разделения пробелами): сама команда и её суффикс.

Базовые арифметические операции

- `AND regd, rega, argb` // regd ← rega & argb
- `EOR regd, rega, argb` // regd ← rega ^ argb
- `SUB regd, rega, argb` // regd ← rega − argb

- `RSB regd, rega, argb` // regd ← argb - rega
- `ADD regd, rega, argb` // regd ← rega + argb
- `ADC regd, rega, argb` // regd ← rega + argb + carry
- `SBC regd, rega, argb` // regd ← rega − argb − !carry
- `RSC regd, rega, argb` // regd ← argb − rega − !carry
- `TST rega, argb` // set flags for rega & argb
- `TEQ rega, argb` // set flags for rega ^ argb
- `CMP rega, argb` // set flags for rega − argb
- `CMN rega, argb` // set flags for rega + argb
- `ORR regd, rega, argb` // regd ← rega | argb
- `MOV regd, arg` // regd ← arg
- `BIC regd, rega, argb` // regd ← rega & ~argb
- `MVN regd, arg` // regd ← ~argb

Суффиксы-условия

```
 EQ        equal  (Z)
NE        not equal  (!Z)
CS or HS  carry set / unsigned higher or same  (C)
CC or LO  carry clear / unsigned lower  (!C)
MI        minus / negative  (N)
PL        plus / positive or zero  (!N)
VS        overflow set  (V)
VC        overflow clear  (!V)
HI        unsigned higher  (C && !Z)
LS        unsigned lower or same  (!C || Z)
GE        signed greater than or equal  (N == V)
LT        signed less than  (N != V)
GT        signed greater than  (!Z && (N == V))
LE        signed less than or equal  (Z || (N != V))
```

Переходы

Счетчик `pc` автоматически увеличивается на 4 при выполнении очередной инструкции. Для ветвления программ изпользуются команды:

- `B label` - переход на метку; используется внутри функций для ветвлений, связанных с циклами или условиями
- `BL label` - сохранение текущего `pc` в `lr` и переход на `label`; обычно используется для вызова функций
- `BX register` - переход к адресу, указанному в регистре; обычно используется для выхода из функций.

Работа с памятью

Процессор может выполнять операции только над регистрами. Для взаимодействия с памятью используются отдельные инструкции загрузки/сохранения регистров.

- `LDR regd, [regaddr]` - загружает машинное слово из памяти по адресу, хранящимся в regaddr, и сохраняет его в регистре regd
- `STR reds, [regaddr]` - сохраняет в памяти машинное слово из регистра regs по адресу, указанному в регистре regaddr.

Адресация данных в памяти и использование библиотечных функций

- Reference по ARM

Основные команды

Как свойственно классической RISC-архитектуре, процессор ARM может выполнять операции только над регистрами. Для доступа к памяти используются отдельные команды *загрузки* (`ldr`) и *сохранения* (`str`).

Общий вид команд:

```
LDR{условие}{тип} Регистр, Адрес
STR{условие]{тип} Регистр, Адрес
```

где `{условие}` - это условие выполнения команды, может быть пустым (см. предыдущий семинар); `{тип}` - тип данных: * `B` - беззнаковый байт * `SB` - знаковый байт * `H` - полуслово (16 бит) * `SB` - знаковое полуслово * `D` - двойное слово.

Если тип в названии команды не указан, то подразумевается обычное слово. Обратите внимание, что для выполнения операций загрузки/сохранения данных, меньших, чем машинное слово, отдельно выделяются знаковые команды, которые делают аккуратное расширение бит нулями, сохраняя при этом старший знаковый бит.

В случае операций загрузки/сохранения пары регистров (двойное слово), регистр должен быть с четным номером. Второе машинное слово подразумевается в соседнем регистре с номером `Rn+1` .

Адресация

Адрес имеет вид: `[R_base {, offset}]` где `R_base` - имя регистра, который содержит базовый адрес в памяти, а необязательный параметр `offset` - смещение относительно адреса. Итоговый адрес определяется как `*R_base + offset` .

Смещение может быть как именем регистра, так и численной константой, закодированной в команду. Регистры обычно используются для индексации элементов массива, константы - для доступа к полям структуры или локальным переменным и аргументам относительно `[sp]` .

Адресация полей Си-структур

По стандарту языка Си, поля в памяти структур размещаются по следующим правилам: * порядок полей в памяти соответствует порядку полей в описании структуры * размер структуры должен быть кратен размеру машинного слова * данные внутри машинных слов размещаются таким образом, чтобы быть прижатыми к их границам.

Таким образом, размер структуры не всегда совпадает с суммой размеров отдельных полей. Например:

```
struct A {
  char  f1; // 1 байт
  int   f2; // 4 байта
  char  f3; // 1 байт
};
// 1 + 4 + 1      = 6 байт
// size(struct A) = 12 байт
```

В данном примере поле `f1` занимает часть машинного слова, поле `f2` - имеет размер 4 байта, поэтому занимает уже следующее машинное слово, и для поля `f3` приходится использовать ещё одно. Простая перестановка полей местами позволяет сэкономить 4 байта:

```
struct A {
  char  f1; // 1 байт
  char  f3; // 1 байт
  int   f2; // 4 байта
};
// 1 + 1 + 4      = 6 байт
// size(struct A) = 8 байт
```

В этом случае поля `f1` и `f3` занимают одно и то же машинное слово.

Компилятор GCC имеет нестандартный аттрибут `packed` , позволяющий создавать "упакованные" структуры, размер которых равен сумме размеров отдельных его полей:

```
struct A {
  char  f1; // 1 байт
  int   f2; // 4 байта
  char  f3; // 1 байт
} __attribute__((packed));
// 1 + 4 + 1      = 6 байт
// size(struct A) = 6 байт
```

Функции стандартной Си-библиотеки

С каждой функцией, которую можно использовать извне, связана некоторая текстовая метка в таблице символов. После компиляции, запись в таблице символов определяет место в памяти, где размещается первая инструкция функции.

Функции, реализованные в разных объектных модулях, но компонуемые в один исполняемый файл, вызываются обычным образом. Способ их вызова ничем не отличается от вызова функций из одного и того же объектного модуля.

При использовании *библиотек*, они загружаются в отдельную область памяти, и на этапе компоновки адрес размещения библиотек не известен.

Более того, размещение самой программы, в общем случае, также предполагается неизвестным.

Такие функции, которые находятся в динамически загружаемых бибилиотеках, включая стандартную библиотеку Си, отображаются в таблице символов с пометкой `@plt`. Их реализация выглядит на языке ассемблера примерно следующим образом:

```
function@plt:

   // Во временный регистр IP загружаем текущий PC
   // с некоторым смещением. По этому смещению находится
   // таблица адресов реальных функций, которая заполняется
   // на этапе загрузки программы и динамических библиотек
   add  ip, pc, #0
   add  ip, ip, #OFFSET_TO_TABLE_BEGIN

   // Загружаем значение адреса из этой таблицы в PC.
   // Это приводит к тому, что переходим к выполнению
   // реальной функции.
   ldr  pc, [ip, #OFFSET_TO_FUNCTION_INDEX]
```

Таким образом, функции из внешних библиотек разполагаются как бы в самой программе, но представляют собой "трамплин" для выполнения реальной функций.

Ассемблер архитектуры x86 (32-bit, и немного про 64-bit)

Основной reference по набору команд преобразованный в HTML.

Reference по наборам команд MMX, SSE и AVX на сайте Intel.

Неплохой учебник по ассемблеру x86 на WikiBooks

32-разрядный ассемблер в 64-битных системах

Мы будем использовать 32-разрядный набор инструкций. На 64-битных архитектурах для этого используется опция компилятора gcc `-m32`.

Кроме того, необходимо установить стек 32-разрядных библиотек. В Ubuntu это делается всего одной командой:

```
sudo apt-get install gcc-multilib
```

Синтаксис AT&T и Intel

Исторически сложилось два синтаксиса языка ассемблера x86: синтаксис AT&T, используемый в UNIX-системах, и синтаксис Intel, используемый в DOS/Windows.

Различие, в первую очередь, относится к порядку аргументов команд.

Компилятор gcc по умолчанию использует синтаксис AT&T, но с указанием опции `-masm=intel` может переключаться в синтаксис Intel.

Кроме того, можно указать используемый синтаксис первой строкой в тексте самой программы:

```
.intel_syntax noprefix
```

Здесь параметр `noprefix` после `.intel_syntax` указывает на то, что помимо порядка аргументов, соответствующих синтаксису Intel, ещё и имена регистров не должны начинаться с символа `%`, а константы - с символа `$`, как это принято в синтаксисе AT&T.

Мы будем использовать именно этот синтаксис, поскольку с его использованием написано большинство доступной документации и примеров, включая документацию от производителей процессоров.

Регистры процессора общего назначения

Исторически семество процессоров x86 унаследовало набор 8-битных регистров общего назначения семества 8080/8085, которые назывались `a`, `b`, `c` и `d`. Но поскольку процессор 8086 стал 16-битным, то регистры стали назваться `ax`, `bx`, `cx` и `dx`. В 32-битных процессорах они называются `eax`, `ebx`, `ecx` и `edx`, в 64-битных `rax`, `rbx`, `rcx` и `rdx`.

Кроме того, в x86 есть регистры "двойного назначения", которые можно использовать, в том числе, в качестве регистров общего назначения, если пользоваться ограниченным подмножеством команд процессора:

- `ebp` - верхняя граница стека;
- `esi` - индекс элемента массива, из которого выполняется копирование;
- `edi` - индекс элемента массива, в который выполняется копирование.

Регистр `esp` содержит указатель на нижнюю границу стека, поэтому произвольным образом его использовать не рекомендуется.

Регистры x86-64

64-разрядные регистры для архитектуры x86-64 именуются начиная с буквы `r`. Помимо регистров `rax` ... `rsi`, `rdi` можно использовать регистры общего назначение `r9` ... `r15`. Указатель стека хранится в `rsp`, верхняя граница стекового фрейма - в `rbp`.

Младшие 32-разрядные части регистров `rax` ... `rsi`, `rdi`, `rsp`, `rbp` можно адресовать по именам `eax` ... `esi`, `edi`, `esp`, `ebp`. При записи значений по 32-битным именам регистров, старшие 32 разряда обнуляются, что приемлемо для операций над 32-разрядными беззнаковыми значениями.

Для работы со знаковыми 32-разрядными значениями, например типом `int`, необходимо предварительно выполнять операции *знакового расширения* с помощью команды `movslq`

Некоторые инструкции

Для синтаксиса Intel первым аргументов команды является тот, значение которого будет модифицировано, а вторым - которое остается неизменным.

```
add     DST, SRC        /* DST += SRC */
sub     DST, SRC        /* DST -= SRC */
inc     DST             /* ++DST */
dec     DST             /* --DST */
neg     DST             /* DST = -DST */
mov     DST, SRC        /* DST = SRC */
imul    SRC             /* (eax,edx) = eax * SRC - знаковое */
mul     SRC             /* (eax,edx) = eax * SRC - беззнаковое */
and     DST, SRC        /* DST &= SRC */
or      DST, SRC        /* DST |= SRC */
xor     DST, SRC        /* DST ^= SRC */
not     DST             /* DST = ~DST */
cmp     DST, SRC        /* DST - SRC, результат не сохраняется, */
test    DST, SRC        /* DST & SRC, результат не сохраняется  */
adc     DST, SRC        /* DST += SRC + CF */
sbb     DST, SRC        /* DST -= SRC - CF */
```

Для синтаксиса AT&T порядок аргументов - противоположный, то есть команда `add %eax, %ebx` вычислит сумму `%eax` и `%ebx`, после чего сохранит результат в регистр `%ebx`, который указан вторым аргументом.

Флаги процессора

В отличии от процессоров ARM, где обновление регистра флагов производится только при наличии специального флага в команде, обозначаемого суффиксом `s`, в процессорах Intel флаги обновляются всегда большинстом инструкций.

Флаг `ZF` устанавливается, если в результате операции был получен нуль.

Флаг `SF` устанавливается, если в результате операции было получено отрицательное число.

Флаг `CF` устанавливается, если в результате выполнения операции произошел перенос из старшего бита результата. Например, для сложения `CF` устанавливается если результат сложения двух беззнаковых чисел не может быть представлен 32-битным беззнаковым числом.

Флаг `OF` устанавливается, если в результате выполняния операции произошло переполнение знакового результата. Например, при сложении `OF` устанавливается, если результат сложения двух знаковых чисел не может быть представлен 32-битным знаковым числом.

Обратите внимание, что и сложение `add`, и вычитание `sub` устанавливают одновременно и флаг `CF`, и флаг `OF`. Сложение и вычитание знаковых и беззнаковых чисел выполняется совершенно одинаково, и поэтому используется одна инструкция и для знаковой, и для беззнаковой операции.

Инструкции `test` и `cmp` не сохраняют результат, а только меняют флаги.

Управление ходом программы

Безусловный переход выполняется с помощью инструкции `jmp`

```
jmp label
```

Условные переходы проверяют комбинации арифметических флагов:

```
jz      label   /* переход, если равно (нуль), ZF == 1 */
jnz     label   /* переход, если не равно (не нуль), ZF == 0 */
jc      label   /* переход, если CF == 1 */
jnc     label   /* переход, если CF == 0 */
jo      label   /* переход, если OF == 1 */
jno     label   /* переход, если OF == 0 */
jg      label   /* переход, если больше для знаковых чисел */
jge     label   /* переход, если >= для знаковых чисел */
jl      label   /* переход, если < для знаковых чисел */
jle     label   /* переход, если <= для знаковых чисел */
ja      label   /* переход, если > для беззнаковых чисел */
jae     label   /* переход, если >= (беззнаковый) */
jb      label   /* переход, если < (беззнаковый) */
jbe     label   /* переход, если <= (беззнаковый) */
```

Вызов функции и возврат из неё осуществляются командами `call` и `ret`

```
call    label   /* складывает в стек адрес возврата, и переход на label */
ret             /* вытаскивает из стека адрес возврата и переходит к нему */
```

Кроме того, есть составная команда для организации циклов, которая подразумевает, что в регистре `ecx` находится счётчик цикла:

```
loop    label   /* уменьшает значение ecx на 1; если ecx==0, то
                   переход на следующую инструкцию, в противном случае
                   переход на label */
```

Адресация памяти

В отличии от RISC-процессоров, x86 позволяет использовать в качестве **один из аргументов** команды как адрес в памяти.

В синтаксисе AT&T такая адресация записывается в виде: `OFFSET(BASE, INDEX, SCALE)`, где `OFFSET` - это константа, `BASE` и `INDEX` - регистры, а `SCALE` - одно из значений: `1`, `2`, `4` или `8`.

Адрес в памяти вычисляется как `OFFSET+BASE+INDEX*SCALE`. Параметры `OFFSET`, `INDEX` и `SCALE` являются опциональными. При их отсутсвтвии подразумевается, что `OFFSET=0`, `INDEX=0`, `SCALE` равен размеру машинного слова.

В синтаксисе Intel используется более очевидная нотация: `[BASE + INDEX * SCALE + OFFSET]`.

Соглашения о вызовах для 32-разрядной архитектуры

Возвращаемое значение 32-разрядного типа функции записывается в регистр `eax`, для возврата 64-разрядного значения используется пара `eax` и `edx`.

Вызываемая функция обязана сохранять на стеке значения регистров общего назначения `ebx`, `ebp`, `esi` и `edi`.

Аргументы могут передаваться в функцию различными способами, в зависимости от соглашений, принятых в ABI.

Соглашения cdecl и stdcall

Соглашения о передаче аргументов, используемые на 32-разрядных системах архитектуры x86. Все аргументы функций складываются справа-налево в стек, затем вызывается функция, которая адресует аргументы через указатель `ebp` или `esp` с некоторым положительным смещением.

Пример:

```
 char * s = "Name";
int value1 = 123;
double value2 = 3.14159;
printf("Hello, %s! Val1 = %d, val2 = %g\n", s, value1, value2);
```

Здесь перед вызовом `printf` в стек будут сложены значения, переменных, прежде чем вызвана функция:

```
 push    value2
push    value1
push    s
push    .FormatString
call    printf
```

В случае использования соглашения `stdcall`, **вызываемая** функция обязана удалить из стека переданные её аргументы после их использования.

В случае использования соглашения `cdecl`, **вызывающая** функция обязана удалить из стека те переменные, которые были переданы в вызываемую функцию.

На языках Си/C++ используемые соглашения можно указывать в специкаторах функций, например:

```
 void __cdecl regular_function(int arg1, int arg2);
#define WINAPI __stdcall
void WINAPI  winapi_function(int arg1, int arg2);
```

Соглашение `stdcall` сейчас используется в основном в операционной системе Windows для обращения к функциям WinAPI. Во всех остальных случаях на 32-разрядных системах используется `cdecl`.

Соглашение fastcall

Если требуется передать в функцию немного целочисленных аргументов, то можно использовать регистры, как в архитектуре ARM. Такое соглашение называется `fastcall`.

Соглашение `fastcall` используется для вызова функций ядра (системных вызовов) в UNIX-подобных системах. В частности, в Linux регистр `eax` используется для передачи номера системного вызова, а регистры `ebx`, `ecx` и `edx` - для передачи целочисленных аргументов.

Аналогичный подход используется и в архитектуре x86-64, где доступных регистров больше, чем в 32-разрядной архитектуре x86.

Соглашения о вызовах для 64-разрядной архитектуры SystemV AMD64 ABI

Целочисленные аргументы передаются последовательно в регистрах: `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`. Если передается более 6 аргументов, то оставшиеся - через стек.

Вещественные аргументы передаются через регистры `xmm0`... `xmm7`.

Возвращаемое значение целочисленного типа должно быть сохранено в `rax`, вещественного - в `xmm0`.

Вызываемая функция обязана сохранять на стеке значения регистров общего назначения `rbx`, `rbp`, и регистры `r12`... `r15`.

Кроме того, при вызове функции для 64-разрядной архитектуры есть дополнительное требование - перед вызовом функции стек должен быть выровнен по границе 16 байт, то есть необходимо уменьшить значение `rsp` таким образом, оно было кратно 16. Если кроме регистров задействуется стек для передачи параметров, то они должны быть прижаты к нижней выровненной границе стека.

Для функций гарантируется 128-байтная "красная зона" в стеке ниже регистра `rsp` - область, которая не будет затронута внешним событием, например, обработчиком сигнала. Таким образом, можно задействовать для адресации локальных переменных память до `rsp-128`.

Вещественная арифметика на x86

Основной reference по набору команд преобразованный в HTML.

Reference по наборам команд MMX, SSE и AVX на сайте Intel.

Сопроцессор x87

Операции над вещественными числами выполняются отдельными блоками процессора. Исторически сложилось, что для вещественной арифметики использовался отдельный *сопроцессор*, а начиная с процессоров 486 (1991 год) этот сопроцессор был интегрирован в кристалл основного процессора.

Таким образом, в целях совместимости со старым кодом, выполнение вещественных операций над числами с плавающей точкой выполняется из предположения наличия сопроцессора.

Компилятор `gcc` использует по умолчанию именно этот способ работы с вещественными числами для 32-разрядной архитектуры (эквивалентно опции `-mfpmath=387`). Для 64-разрядной архитектуры по умолчанию используется набор команд SSE (`-mfpmath=sse`).

Взаимодействие с сопроцессором x87 организовано в форме записи операндов в стек и выполнения операций над элементами этого стека. Команды сопроцессора обычно начинаются с буквы `f`, и оперируют с регистрами, которые обозначаются от `st(0)` (вершина стека) до `st(7)` (последний регистр FPU).

Выполнять арифметические FPU-операции можно над любыми регистрами этого стека, но операции, позволяющие взаимодействие с памятью, возможны только через вершину стека `st(0)`.

Основные инструкции x86:

```
fld SIZE ptr ADDR // загрузить в стек значение из памяти
fld REG           // поместить на вершину стека значение из другого регистра st(X)
fld1              // поместить на вершину стека значение 1
fldz              // 0
fldpi             // π
fst SIZE ptr ADDR // сохранить из стека в память

fild SIZE ptr ADDR // загрузить целое число в стек
fist SIZE ptr ADDR // сохранить целое число из стека

fcom              // сравнение st(0) с памятью
fcomi             // сравнение st(0) с st(i)

fadd
fsub
fmul
fdiv              // операции над вещественными операндами

fiadd
fisub
fimul
fidiv             // операции над целочисленными операндами
```

Регистры MMX/SSE/AVX/AVX-512

В современных Intel/AMD процессорах есть 16 регистров (в 32-разрядном режиме доступны только 8), которые предназначены как для вещественных операций, так и для целочисленных.

128-битные регистры MMX/SSE именуются `xmm0` ... `xmm7`, `xmm8` ... `xmm15`.

256-битные регистры AVX `ymm0` ... `ymm15` подразумевают, что их младшие 128 бит совпадают с регистрами MMX/SSE.

512-битные регистры AVX-512 (новые Xeon и Core i9) `zmm0` ... `zmm15` подразумевают, что младшие 256 бит совпадают с регистрами AVX.

Скалярные инструкции SSE

Несмотря на свой большой размер, регистры SSE можно использовать как обычные скалярные, что намного эффективнее, чем x87 FPU.

В отличии от регистров в стеке `st(0)` ... `st(7)`, все регистры SSE являются равнозначными.

```
// Копирование регистр-регистр и регистр-память
movsd   DST, SRC  // пересылка double
movss   DST, SRC  // пересылка float

// Арифметические
addsd   DST, SRC   // DST += SRC, double
addss   DST, SRC   // DST += SRC, float
subsd   DST, SRC   // DST -= SRC, double
subss   DST, SRC   // DST -= SRC, float
mulsd   DST, SRC   // DST *= SRC, double
mulss   DST, SRC   // DST *= SRC, float
divsd   DST, SRC   // DST /= SRC, double
divss   DST, SRC   // DST /= SRC, float
sqrtsd  DST, SRC   // DST = sqrt(SRC), double
sqrtss  DST, SRC   // DST = sqrt(SRC), float
maxsd   DST, SRC   // DST = max(DST, SRC), double
maxss   DST, SRC   // DST = max(DST, SRC), float
minsd   DST, SRC   // DST = min(DST, SRC), double
minss   DST, SRC   // DST = min(DST, SRC), float

// Преобразования
cvtsd2si DST, SRC  // double -> int
cvtsi2sd DST, SRC  // int -> double

// Сравнения (операция DST-SRC, которая меняет флаги)
comisd  DST, SRC  // для double
comiss  DST, SRC  // для float
```

Соглашения о вызовах для x86 (32 бит)

Все вещественные аргументы передаются в функцию через стек. Возвращаемое вещественное значение должно быть сохранено в регистре `st(0)`, причем это требование необходимо соблюдать **даже при использовании регистров SSE**. Это необходимо для того, чтобы вызывающая функция могла использовать вещественный результат, независимо от способа реализации вызываемой функции.

Переместить результат из регистра SSE в регистр x87 можно только с ипользованием памяти:

```
sub     esp, 8          // выделяем 8 байт на стеке
movsd   [esp], xmm0     // копируем из xmm0 в память
fld     qword ptr [esp] // загружаем из памяти в стек x87
add     esp, 8          // освобождаем память на стеке
```

Векторные инструкции SSE и intrisics-функции на Си

Между регистрами можно выполнять *векторные* операции, то есть операции сразу над несколькими 8, 16, 32 или 64-битными значениями, которые хранятся в паре 128-битных регистров.

Общий вид таких команд следующий:

```
OPERATION p [s|d]
```

где `OPERATION` - это одна из операций `add`, `mul` и т.д., буква `p` в названии команды является сокращением от `p`acked, а `s` или `d` - это `s`ingle или `d`ouble точность вещественных чисел.

Загрузка/сохранение выполняется вариантами команды `mov`:

```
mov[ap|up][s|d]   DST, SRC
```

где `ap` - загрузка/сохнанение из памяти, выровненной по границе размера регистра (16 байт), `up` - для невыровненной памяти.

Использование операндов в памяти для операций, отличных от `mov`, возможно только для выровненной памяти.

Для задействования векторных инструкций не обязательно использовать язык ассемблера. Компиляторы Intel и `gcc` имеют поддержку псевдо-функций, объявленных в заголовочных файлах вида `*intrin.h`, которые транслируются в эти инструкции при компиляции. Подробный Reference доступен здесь.

Жизнь без стандартной библиотеки

Основной reference по набору команд преобразованный в HTML.

Инструменты для сборки без стандартной библиотеки

Инструменты GNU

При компоновке с опцией `-nostdlib` линковщик не включает функцию `main`, и не связывает программу со стандартной библиотекой языка Си.

Получаемый на выходе файл - обычный выполняемый файл в формате ELF, который можно выполнить в операционной системе.

Размещение различных секций файла при компоновке можно указать в специальном ld-файле (подробнее см. LD: Scripts), который указывается опцией `-T имя_файла`.

Для того, чтобы при компоновке не включалась лишняя информация о том, каким компилятором собрана программа, исползуется опция линковщика `--build-id=none`.

Для выделения кода самой программы из ELF-файла можно использовать утилиту `objcopy`.

Ассемблер NASM

Ассемблер `nasm` использует хоть и похожий на Intel, но всё же немного отличающийся по синтаксису язык. Этот ассемблер, в отличии от GNU, поддерживает много выходных форматов, в том числе flat-файлы, предназначенные для непосредственной заливки программатором или загрузки в память.

Взаимодействие с внешним миром в Linux

Общие сведения о системных вызовах

Системные вызовы - это функции, реализованные в ядре операционной системы, и поэтому обычные процессы могут вызывать их только используя специальные команды, которые переключают процессор в режим ядра. Для доступа к системным вызовам используются нестандартные способы вызова: либо механизм прерываний (команда `int`), либо специализированная команда архитектуры x86-64 `syscall`.

Для большинства (но не для всех) системных вызовов реализованы Си-сигнатуры, которые описаны во 2-м разделе man-страниц. Поскольку соглашения о вызовах обычных Си-функций отличаются от соглашений о системных вызовах, стандартная библиотека языка Си содержит короткие функции-оболочки, единственная задача которых - это переложить аргументы в соотвествии с требуемым соглашением, после чего выполнить системный вызов, и вернуть результат.

Примеры некоторых системных вызовов в Linux:

- `exit` (`_exit` в Си-нотации) = `1` - выход из программы;
- `read` = `3` - чтение из файлового дескриптора;
- `write` = `4` - запись в файловый дескриптор;
- `brk` (`sbrk` в Си-нотации) = `45` - перемещение границы сегмента данных программы.

Для обращения к произвольному системному вызову по его номеру, например, если для него не реализована функция-оболочка в стандартной Си-библиотеке, используется функция `syscall`:

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

int main()
{
    const char Hello[] = "Hello!\n";

    // эквивалентно вызову
    // write(1, Hello, sizeof(Hello)-1);
    syscall(SYS_write, 1, Hello, sizeof(Hello)-1);
}
```

32-разрядные системы x86

Операционная система Linux реализует системные вызовы через программное прерывание с номером `0x80`, которое можно инициировать командой `int`. В регистре `eax` хранится номер системного вызова, в регистрах `ebx`, `ecx`, `edx`, `esi`, `edi` передаются аргументы, а возвращаемое значение передается через `eax`.

Номера системных вызовов на x86 перечислены в файле `/usr/include/asm/unistd_32.h`.

Пример для x86 (вывод строки `Hello` с использованием системного вызова `write`):

```
    .text
    ......
    mov   eax, 4  // 4 - номер write
    mov   ebx, 1  // 1 - файловый дескриптор stdout
    mov   ecx, hello_ptr // указатель на hello
    mov   edx, 5  // количество байт в выводе
    int   0x80    // системный вызов Linux
    ......
    .data
hello:
    .string "Hello"
hello_ptr:
    .long   hello
```

64-разрядные системы x86-64

В 64-битных системах возможно использовать соглашения о системных вызовах для 32-битных платформ x86, но этот механизм используется исключительно для обеспечения работоспособности старых 32-битных программ. При использовании инструкции `int 0x80` аргументы, передаваемые через регистры, усекаются до 32-битных значений, что может приводить к неопределенному поведению, например, если передаются указатели.

```
// переменная хранится на стеке, поэтому ее адрес
// имеет достаточно большое значение в виртуальном
// 64-разрядном адресном пространстве процесса
char buffer[1024];

// если использовать int 0x80, значение указателя buffer
// будет записано в 32-битный регистр ecx, что приведет к
// ошибке Segmentation Fault
ssize_t bytes_read = read(0, buffer, sizeof(buffer));
```

Родным для архитектуры x86-64 соглашением в Linux является использование команды процессора `syscall`, где номер системного вызова передается через `rax`, а аргументы передаются через регистры: `rdi`, `rsi`, `rdx`, `r10`, `r8` и `r9`. Обратите внимание, что не все используемые регистры совпадают со стандартным соглашением о вызовах в x86-64, например, вместо регистра `rcx` используется регистр `r10`. Кроме того, использование команды `syscall` может испортить содержимое регистров `rcx` и `r11`.

Номера системных вызовов для использования их командой `syscall`, хранятся в заголовочном файле `/usr/include/sys/syscall.h`, и большинство из них совпадают (хотя это ничем не гарантируется) с номерами системных вызовов для 32-битных системных вызовов архитектуры x86.

Пример для x86-64 (вывод строки `Hello` с использованием системного вызова `write`):

```
    .text
    ......
    mov   rax, 4  // 4 - номер write
    mov   rdi, 1  // 1 - файловый дескриптор stdout
    mov   rsi, hello_ptr // указатель на hello
    mov   rdx, 5  // количество байт в выводе
    syscall       // системный вызов Linux
    ......
    .data
hello:
    .string "Hello"
hello_ptr:
    .quad   hello
```

Взаимодействие с внешним миром через BIOS или в DOS (историческая справка)

До момента загрузки операционной системы, обработка ввода-вывода осуществляется с помощью подпрограмм, предоставляемых BIOS (Basic Input Output System).

Разные подсистемам ("сервисам") соответствуют различные номера прерываний. Например, прерывание `0x10` предназначено для вывода на экран, а прерывание `0x09` - за чтение с клавиатуры.

Некоторые операционные системы, например DOS, не запрещают использование прерываний BIOS, а дополняют их своими механизмами.

Подробное описание функций BIOS и DOS - здесь.

Отдельно стоит рассмотреть взаимодействие с выводом на экран. Поскольку вывод через прерывание является хоть и универсальным, но все же медленным способом, то лучше использовать прямую запись в видеопамять VGA.

Видеопамять VGA в архитектуре x86 располагается в диапазоне `0xA000..0xDFFFF` (256Кб начиная с 640Кб), и делится на "окна", - области, назначение которых зависит от используемого режима работы.

В стандартном текстовом видеорежиме, вывод символа в позицию `(X, Y)` осуществляется записью двух байт по адресу `0xB8000+Y*80*2+X*2`, где младший байт означает код символа, а старший - цвет символа и фона.

Стадии загрузки системы

Включение компьютера

Сразу после запуска компьютера, управление передаётся программе из ROM-памяти (часто именуемую BIOS, хотя это не совсем корректно), задача которой - выполнить диагностику системы, определить конфигурацию оборудования, и загрузить программу-загрузчик с определенного диска, чтобы передать ей управление.

Программа-загрузчик может располагаться: * в классической PC-системе - в первых 512 байтах диска; * в современных системах с EFI/UEFI - выделяется определенная область в Flash-памяти на системной плате, куда установщик операционной системы записывает свой загрузчик.

Загрузка системы через MBR

Master Boot Record имеет размер 512 байт, и состоит из двух частей: программы-загрузчика и первичной таблицы разделов диска. Признаком того, что MBR имеет загрузчик, является значение `0x55AA` в последних двух байтах. Размер первичной таблицы разделов для PC - 64 байта, таким образом, для загрузчика остается всего 446 байт (512-2-64).

Если загрузчик является достаточно сложным (например, GRUB в графическом режиме со всякими красивостями и умной командной строкой), то его делят на две части: в MBR и частично - на разделе диска.

Первые 446 байт загружаются с диска в память по адресу `0x7C00`, а область памяти от `0x0000` до `0x7C00` считается зарезервированной под стек. При этом, процессор x86 работает в 16-битном реальном режиме, со старинной сегментной адресацией памяти. Пример программирования MBR - здесь.

Задача загрузчика - это найти на диске файл с *ядром* системы, загрузить его в память, и передать ему управление. Примеры файлов ядра: * `C:\msdos.sys` - для DOS; * `C:\Windows\System32\ntoskrnl.exe` - для Windows; * `/boot/vmlinuz` - символическая ссылка на zlib-сжатый образ ядра в Linux.

Формат файла ядра - как правило, соответствует обычному исполняемому файлу (PE для Windows или ELF для Linux), но на него накладываются некоторые ограничения о размещении данных внутри файла, и кроме того, этот файл не может иметь зависимости от каких-либо библиотек.

Загрузка ядра Linux, формат `multiboot`.

Загрузчик GRUB загружает ELF-файл с ядром, распаковывает его при необходимости, и размещает по адресу, начиная с 1Мб.

Далее загрузчик ищет Magic-метку заголовка `multiboot` в первых 32K загруженного файла ядра, сразу после которой идет набор флагов и контрольная сумма заголовка. После этого заголовка, в самом файле следует 16K памяти под стек, а сразу после него - начало программы, которую нужно выполнять.

Таким образом, при компиляции ядер, необходимо строго указывать очерёдность различных секций, чтобы GRUB смог запустить ядро.

Подробнее - здесь.

Запуск ядра

Ядро регистрирует вектор прерываний, выполняет дальнейшую инициализацию оборудования, загружая при необходимости различные драйверы устройств. Когда ядро полностью загружено, то выполняется загрузка первой программы, которая выполняется в режиме пользователя: * `C:\command.com` - для DOS; * `C:\Windows\System32\smms.exe` - для Windows; * `/boot/initrd` - для Linux.

Дальнейшие стадии запуска

Процесс `initrd`, в зависимости от дистрибутива: * классический Unix-way: запускает набор shell-скриптов в одном из подкаталогов `/etc/init.d/rcX.d`, где `X` - уровень запуска по умолчанию, прописанный в файле `/etc/inittab`; * SystemD-way: запускает программу `systemd`, которая имеет свой набор конфигурационных файлов, по которым строит дерево зависимостей различных служб, и запускает их.

Файловый ввод-вывод

Файловые дескрипторы

Файловые дескрипторы - это целые числа, однозначно идентифицирующие открытые файлы в рамках одной программы.

Как правило, при запуске процесса дескрипторы `0`, `1` и `2` уже заняты стандартным потоком ввода (`stdin`), стандартным потоком вывода (`stdout`) и стандартным потоком ошибок (`stderr`).

Файловые дескрипторы могут быть созданы с помощью операции создания или открытия файла.

Системные вызовы open/close

Системный вызов `open` предназначен для создания файлового дескриптора из существующего файла, и имеет формальную сигнатуру:

```
int open(const char *path, int oflag, ... /* mode_t mode */);
```

Первый параметр - имя файла (полное, или относительно текущего каталога). Второй параметр - параметры открытия файла, третий (опциональный) - права доступа на файл при его создании.

Основные параметры открытия файлов: * `O_RDONLY` - только для чтения; * `O_WRONLY` - только на запись; * `O_RDWR` - чтение и запись; * `O_APPEND` - запись в конец файла; * `O_TRUNC` - обнуление файла при открытии; * `O_CREAT` - создание файла, если не существует; * `O_EXCL` - создание файла только если он не существует.

В случае успеха возвращается неотрицательное число - дескриптор, в случае ошибки - значение `-1`.

Обработка ошибок в POSIX

Код ошибки последней операции хранится в глобальной целочисленной "переменной" `errno` (на самом деле, в современных реализациях - это макрос). Значения кодов можно определить из `man`-страниц, либо вывести текст ошибки с помощью функции `perror`.

Аттрибуты файлов в POSIX

В случае создания файла, обязательным параметром является набор POSIX-аттрибутов доступа к файлу. Как правило, они кодируются в восьмиричной системе исчисления в виде `0ugo`, где `u` - права доступа для владельца файла, `g` - права доступа для всех пользователей группы файла, `o` - для остальных.

В восьмеричной записи значения от 0 до 7 соответствуют комбинации трёх бит:

```
00: ---
01: --x
02: -w-
03: -wx
04: r--
05: r-x
06: rw-
07: rwx
```

Чтение и запись в POSIX

Чтение и запись осуществляются с помощью системных вызовов:

```
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

Здесь `buf` - указатель на буфер данных, а `count` - максимальное количество байт для чтения/записи.

Как правило, в `count` указывается размер буфера данных при чтении, или количество данных при записи.

Возвращаемый тип `ssize_t` - целочисленный, определенный в диапазоне `[-1...SSIZE_MAX]`, где `SSIZE_MAX` обычно совпадает с `SIZE_MAX/2`. Значение `-1` используется в качестве признака ошибки, неотрицательные значения - количество записанных/прочитанных байт, которое может быть меньше, чем `count`.

Если системный вызов `read` вернул значение `0`, то достигнут конец файла, либо был закрыт канал ввода.

Навигация по файлу в POSIX

Если файл является обычным, то можно выполнять перемещение текущей позиции в файле.

```
off_t lseek(int fd, off_t offset, int whence);
```

Этот системный вызов предназначен для перемещения текущего указателя на файл.

Третий параметр `whence` один из трех стандартных способов перемещения: * `SEEK_SET` - указать явным образом позицию в файле; * `SEEK_CUR` - сместить указатель на определенное смещение относительно текущей позиции; * `SEEK_END` - сместить указатель на определенное смещение относительно конца файла.

Системный вызов `lseek` возвращает текущую позицию в файле, либо значение `-1` в случае возникновения ошибки.

Тип `off_t` является знаковым, и по умолчанию 32-разрядным. Для того, чтобы уметь работать с файлами размером больше 2-х гигабайт, определяется значение переменной препроцессора **до подключения заголовочных файлов**:

```
#define _FILE_OFFSET_BITS 64
```

В этом случае, тип данных `off_t` становится 64-разрядным. Определить значение переменных препроцессора можно не меняя исходные тексты программы, передав компилятору опцию `-ДКЛЮЧ=ЗНАЧЕНИЕ`:

```
# Скомпилировать программу с поддержкой больших файлов
gcc -D_FILE_OFFSET_BITS=64 legacy_source.c
```

Компиляция и запуск Windows-программ из Linux

Для кросс-компиляции используется компилятор gcc с целевой системой `w64-mingw`. Устанавливается из пакета: * `mingw32-gcc` - для Fedora * `gcc-mingw-w64` - для Ubuntu * `mingw32-cross-gcc` - для openSUSE.

Скомпилировать программу для Windows можно командой:

```
$ i686-w64-mingw-gcc -m32 program.c
# На выходе получаем файл a.exe, а не a.out
```

Обратите внимание, что система Linux, в отличии от Windows различает регистр букв в файловой системе, поэтому нужно использовать стандартные заголовочные файлы WinAPI в нижнем регистре:

```
#include <windows.h> // правильно
#include <Windows.h> // скомпилируется в Windows, но не в Linux
```

Запустить полученный файл можно с помощью WINE:

```
$ WINEDEBUG=-all wine a.exe
```

Установка переменной окружения `WINEDEBUG` в значение `-all` приводит к тому, что в консоль не будет выводится отладочная информация, связанная с подсистемой `wine`, которая перемешивается с выводом самой программы.

Файловые дескрипторы и другие типы данных в WinAPI

В системе Windows для файловых дескрипторов используется тип `HANDLE`.

Для однобайтных строк используется тип `LPCSTR`, для многобайтных - `LPCWSTR`.

Функции WinAPI, которые имеют разные варианты поддержки строк, и работают с однобайтными функциями, заканчиваются на букву `A`, а функции, которые работают с многобайтными файлами - на букву `W`.

Для беззнакового 32-разрядного числа, используемого для флагов - тип `DWORD`.

Полный список типов данных в документации от Microsoft.

Функции WinAPI для работы с файлами

Файл можно открыть с помощью функции CreateFile.

Чтение и запись - с помощью функций ReadFile и WriteFile.

Навигация по файлу - с помощью функции SetFilePointerEx.

Свойства файлов

Сведения о файле

Структура `stat`

С каждым файлом в файловой системе связана метаинформация (status), которая определяется структурой `struct stat`:

```
struct stat {
   dev_t     st_dev;         /* ID of device containing file */
   ino_t     st_ino;         /* Inode number */
   mode_t    st_mode;        /* File type and mode */
   nlink_t   st_nlink;       /* Number of hard links */
   uid_t     st_uid;         /* User ID of owner */
   gid_t     st_gid;         /* Group ID of owner */
   dev_t     st_rdev;        /* Device ID (if special file) */
   off_t     st_size;        /* Total size, in bytes */
   blksize_t st_blksize;     /* Block size for filesystem I/O */
   blkcnt_t  st_blocks;      /* Number of 512B blocks allocated */

   struct timespec st_atim;  /* Time of last access */
   struct timespec st_mtim;  /* Time of last modification */
   struct timespec st_ctim;  /* Time of last status change */

   /* Backward compatibility */
   #define st_atime st_atim.tv_sec
   #define st_mtime st_mtim.tv_sec
   #define st_ctime st_ctim.tv_sec
};
```

Метаинформацию о файле получается с помощью команды `stat ИМЯ_ФАЙЛА` или одного из системных вызовов: * `int stat(const char *file_name, struct stat *stat_buffer)` - получение информации о файле по его имени; * `int fstat(int fd, struct stat *stat_buffer)` - то же самое, но для открытого файлового дескриптора; * `int lstat(const char *path_name, struct stat *stat_buffer)` - аналогично `stat`, но в случае, если имя файла указывает на символическую ссылку, то возвращается информация о самой ссылке, а не о файле, на который она ссылается.

Режим доступа и типы файлов в POSIX

В POSIX есть несколько основных типов файлов:

- Регулярный файл (`S_IFREG = 0100000`). Занимает место на диске, содержит самые обычные данные.
- Каталог (`S_IFDIR = 0040000`). Файл специального типа, который хранит список имен файлов.
- Символическая ссылка (`S_IFLNK = 0120000`). Файл, который ссылается на другой файл (в том числе в другом каталоге или даже на другой файловой системе), и с точки зрения фукнций ввода-вывода ничем не отличается от того файла, на который он ссылается.
- Блочные (`S_IFBLK = 0060000`) и символьные (`S_IFCHR = 0020000`) устройства. Используются как удобный способ взаимодействия с оборудованием.
- Именованные каналы (`S_IFIFO = 0010000`) и сокеты (`S_IFSOCK = 0140000`), предназначенные для межпроцессного взаимодействия.

Тип файла закодирован в одном поле структуры с режимом доступа (`rwxrwxrwx`) - целочисленном `.st_mode`.

Для выделения отдельных типов файлов применяются поразрядные операции с помощью одного из макросов: `S_ISREG(m)`, `S_ISDIR(m)`, `S_ISCHR(m)`, `S_ISBLK(m)`, `S_ISFIFO(m)`, `S_ISLNK(m)` и `S_ISSOCK(m)`, которые возвращают `0` в качестве false, и произвольное ненулевое значение в качестве true.

Для выделения режима доступа, который закодирован в младших битах `.st_mode`, которые можно извлекать, применяя поразрядные операции с помощью констант `S_IWUSR`, `S_IRGRP`, `S_IXOTH` и др. Полный список констант можно посмотреть в `man 7 inode`.

Доступ к файлу

У каждого файла, помимо режима доступа (`rwx` для владельца, группы и остальных) есть два идентификатора, - целых положительных числа: * `.st_uid` - идентификатор пользователя-владельца файла; * `.st_gid` - идентификатор группы-владельца файла.

Права доступа "для владельца" применяются при совпадении идентификатора текущего пользователя (можно получить с помощью `getuid()`) с полем `.st_uid`. Аналогично - для группы при совпадении `getgid()` с `.st_gid`. В остальных случаях действуют права доступа "для остальных".

Удобным способом определения прав у текущего пользователя является использование системного вызова `access`:

```
int access(const char *path_name, int mode)
```

Этот системный вызов принимает в качестве параметра `mode` поразрядную комбинацию из флагов `R_OK`, `W_OK`, `X_OK` и `F_OK`, - соответственно

способность чтения, записи, выполнения файла, и его существование. Возвращает 0 в случае, если перечисленные аттрибуты допустимы для текущего пользователя, и -1 в противном случае.

Маска создания файлов

При создании новых файлов с помощью системного вызова `open` (и всех высокоуровневых функций, которые используют `open`), нужно обязательно указывать режим доступа для вновь созданных файлов.

В реальности режим доступа может отличаться от запрошенного: для вновь созданного файла (или каталога) применяется *маска создания файлов* с помощью поразрядной операции "и-не":

```
 /* Пусть umask = 0222 */
open("new_file", O_WRONLY|O_CREAT, 0666); // OK

/* Создан файл с аттрибутами 0666 & ~0222 = 0444 */
```

По умолчанию маска создания файлов равна `0000`, то есть не накладывает никаких ограничений. Системный вызов `umask` позволяет задать явным образом новую маску, которая может быть использована для предотвращения случайного создания файлов со слишком слабозащищенными правами доступа.

Ссылки и файловые дескрипторы

Пара значений, хранящихся в полях `.st_dev` и `.st_ino`, позволяют однозначно идентифицировать любой файл в виртуальной файловой системе до её перезагрузки или отмонтирования каких-то частей файловой системы.

Поле `.st_nlink` хранит количество имен, связанных с данным именем файла, причем они могут находиться в разных каталогах одной физической файловой системы. Для файла, который существует в файловой системе, `.st_nlink` всегда не меньше `1`, при удалении файла это число становится равным `0`. Но если файл открыт хотя бы одним процессом в системе, то физически он не удаляется (хотя по имени его уже не найти), и доступен по файловому дескриптору до момента закрытия файла.

Удалить программным спосособом файл можно с помощью системного вызова `unlink`. Как следует из названия, этот системный вызов уменьшает количество ссылок `.st_nlink`.

Для предотвращения *состояния гонки* (race condition), у многих системных вызовов для работы с файлами существуют аналоги, подразумевающие работу с файловыми дескрипторами, а не с именами файлов.

Пример состояния гонки:

```
 struct stat st;
assert(0==stat("my_file.txt", &st)); // OK

/* теперь кто-нибудь извне успевает удалить или
   переименовать файл между моментами этих двух вызовов */

int fd = open("my_file.txt", O_RDONLY); // ERR: файл не найден
/* ??? как же так? Только что был здесь, а теперь недоступен */
```

Если немного переделать этот пример с использованием `fstat`, то проблема решена:

```
 int fd = open("my_file.txt", O_RDONLY);
if (-1!=fd) {

    /* Теперь кто-нибудь удаляет файл, или переименовывает
    его. Но нас это уже не должно волновать: файл существует
    до того момента, пока мы его не закроем */

    struct stat st;
    fstat(fd, &st);
    off_t file_size = st.st_size; // размер доступных данных
}
```

Разные полезные системные вызовы

- Добавление нового имени (увеличение ссылок) - `man 2 link`
- Удаление (уменьшение ссылок) - `man 2 unlink`
- Создание символической ссылки - `man 2 symlink`
- Чтение значения символической ссылки - `man 2 readlink`
- Создание каталога - `man 2 mkdir`
- Удаление пустого каталога - `man 2 rmdir`
- Изменение режима доступа - `man 2 chmod`

- Перемещение (переименование) файла - `man 2 rename`

Аттрибуты файловых дескрипторов

Системный вызов `fcntl` предназначен для управления открытыми файлыми дескрипторами.

```
int fcntl(int fd, int get_command);
int fcntl(int fd, int set_command, void *set_value);
```

Для открытых файлов можно командами `F_GETFL` и `F_SETFL` получать и менять аттрибуты открытия: `O_APPEND`, `O_ASYNC`, `O_NONBLOCK`. В Linux не возможно изменить режим открытия файлов (например поменять `O_RDONLY` на `O_RDRW`), хотя в некоторых UNIX системах это допускается.

Блокировки файлов

Проблема состояния гонки относится не только к возможным изменениям аттрибутов файлов, но и к проблеме *гонки данных* (data race) между различными процессами, которые хотят одновременно обращаться к одним и тем же файлам.

В UNIX-подобных системах существует два основных механизма для блокировки файлов.

Блокировки BSD flock

Поддерживается *BSD и Linux. Системный вызов `flock` имеет сигнатуру:

```
int flock(int fd, int operation);
```

`fd` - открытый файловый дескриптор, возможные операции: * `LOCK_SH` - получить разделяемую блокировку. Разделяемую блокировку на файл могут накладывать разные процессы, не мешая друг другу, когда им требуется только читать данные из файла. * `LOCK_EX` - получить эксклюзивную блокировку. Только один процесс может это сделать. * `LOCK_UN` - разблокировать файла.

Блокировка накладывается не на файловые дескрипторы, а на сами файлы, с с которыми они связаны. При попытке получить блокировку (любого типа) к файлу, который кем-то эксклюзивно заблокирован, приведет к приостановке текущего процесса до тех пора, пока блокировка не будет снята.

Блокировки POSIX file lock

Использует команды `F_GETLK` (получить блокировки), `F_SETLK` (установить блокиовку) и `F_SETLKW` (установить блокировку, но сначала дождаться, когда это возможно) системного вызова `fcntl` для управления блокировками.

В качестве третьего аргумента `fcntl` передается структура:

```
struct flock {
    off_t   l_start;  /* starting offset    */
    off_t   l_len;    /* len = 0    means until end of file */
    pid_t   l_pid;    /* lock owner */
    short   l_type;   /* lock type: read/write, etc. */
    short   l_whence; /* type of    l_start */
    int     l_sysid;  /* remote system id or zero for local */
};
```

В отличии от BSD `flock`, этот способ является более гибким, и позволяет управлять блокировками отдельных частей файла.

Пример:

```
 struct flock fl;
memset(&fl, 0, sizeof(fl));

// Установить блокировку только для чтения (не эксклюзивно)
fl.l_type = F_RDLCK;

// Блокировка на весь файл
fl.l_whence = SEEK_SET; // по аналогии с lseek
fl.l_start = 0;         // по аналогии с lseek
fl.l_len = 0;           // 0 - до конца, либо размер области

// Установить блокировку для чтения. Если какой-то процесс
// уже захватил файл для записи, то ожидание, пока освободится
fcntl(fd, F_SETLKW, &fl);

// Блокировка на запись с 10 по 15 байты файла
fl.l_type = F_WRLCK;
fl.l_start = 10;
fl.l_len = 5;
fcntl(fd, F_SETLK, &fl);

// Снять блокировку с этого же диапазона
fl.l_type = F_UNLCK;
fcntl(fd, F_SETLK, &fl);

// При закрытии файла процессом все блокировки недействительны
close(fd);
```

Блокировка BSD lockf

Упрощенным аналогом для блокировки отдельных частей файла является системный вызов `lockf`; для указания начала области блокировки нужно переместить указатель `lseek`.

```
int lockf(int fd, int cmd, off_t len);
```

В отличии от `fcntl`, этот системный вызов подразумевает только эксклюзивную блокировку (для записи).

Команды flock и lslocks

Команда `flock` позоляет установить блокировку на произвольный файл для запуска какой-либо программы, и снимает блокировку при завершении работы дочернего процесса.

Команда `lslocks` отображает таблицу файлов с блокировками.

Страницы памяти в виртуальном адресном пространстве

Инструменты

Кроме пошагового отладчика `gdb`, при работе с памятью существуют дополнительные инструменты, предназначенные для выявления проблем.

Интерпретируемое выполнение с помощью `valgrind`

Набор инструментов `valgrind` использует контролируемое выполнение инструкций программы, модифицируя её код перед выполнением на физическом процессоре.

Основные инструменты: * `memcheck` - диагностика проблем с памятью: неверные указатели на кучу, повторное освобождение, чтение неинициализированных данных и забытое освобождение памяти. * `callgrind` - диагностика производительности выполняемой программы.

Для запуска программы под valgrind необходимо собрать программу с отладочной информацией (опция компиляции `-g`), в противном случае вывод valgrind будет не информативным.

Запуск:

```
> valgrind --tool=ИНСТРУМЕНТ program.jpg ARG1 ARG2 ... ARGn
```

В случае использования инструмента `callgrind`, после выполнения программы генерируется файл `callgrind.out` в формате XML, который можно визуализировать с помощью KCacheGrind (из KDE во всех современных дистрибутивах Linux), либо его кросс-платформенном аналоге QCacheGrind.

Runtime-проверки ошибок с помощью sanitizer'ов

Требует для своей работы свежие версии `clang` или `gcc`, и позволяют выполнять инструментальный контроль во время выполнения программы значительно быстрее, чем `valgrind`.

Реализуются на уровне генерации кода и замены некоторых функций, например `malloc` / `free` на реализации с дополнительными проверками.

Основные санитайзеры: * AddressSanitizer (`-fsanitize=address`) - диагностирует ситуации утечек памяти, двойного освобождения памяти, выхода за границу стека или кучи, и использования указателей на стек после завершения работы функции. * MemorySanitizer (`-fsanitize=memory`) - диагностика ситуаций чтения неинициализированных данных. Требует, чтобы программа, и как и все используемые ею библиотеки, были скомпилированы в позиционно-независимый код. * UndefinedBehaviourSanitizer (`-fsanitize=undefined`) - диагностика неопределенного поведения в целочисленной арифметике: битовые сдвиги, знаковое переполнение, и т.д.

Системный вызов mmap

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

void *mmap(
    void *addr,    /* рекомендуемый адрес отображения */
    size_t length, /* размер отображения */
    int prot,      /* аттрибуты доступа */
    int flags,     /* флаги совместного отображения */
    int fd,        /* файловый дескриптор фала */
    off_t offset   /* смещение относительно начала файла */
  );

int munmap(void *addr, size_t length) /* освободить отображение */
```

Системный вызов `mmap` предназначен для создания в виртуальном адресном пространстве процесса доступной области по определенному адресу. Эта область может быть как связана с определенным файлом (ранее открытым), так и располагаться в оперативной памяти. Второй способ использования обычно реализуется в функциях `malloc` / `calloc`.

Память можно выделять только постранично. Для большинства архитектур размер одной страницы равен 4Кб, хотя процессоры архитектуры x86_64 поддерживают страницы большего размера: 2Мб и 1Гб.

В общем случае, никогда нельзя полагаться на то, что размер страницы равен 4096 байт. Его можно узнать с помощью команды `getconf` или функции `sysconf`:

```
# Bash
> getconf PAGE_SIZE
4096

/* Си */
#include <unistd.h>
long page_size = sysconf(_SC_PAGE_SIZE);
```

Параметр `offset` (если используется файл) обязан быть кратным размеру страницы; параметр `length` - нет, но ядро системы округляет это значение до размера страницы в большую сторону. Параметр `addr` (рекомендуемый адрес) может быть равным `NULL`, - в этом случае ядро само назначает адрес в виртуальном адресном пространстве.

При использовании отображения на файл, параметр `length` имеет значение длины отображаемых данных; в случае, если размер файла меньше размера страницы, или отображается его последний небольшой фрагмент, то оставшаяся часть страницы заполняется нулями.

Страницы памяти могут флаги аттрибутов доступа: * чтение `PROT_READ`; * запись `PROT_WRITE`; * выполнение `PROT_EXE`; * ничего `PROT_NONE`.

В случае использования отображения на файл, он должен быть открыт на чтение или запись в соответствии с требуемыми аттрибутами доступа.

Флаги `mmap`: * `MAP_FIXED` - требует, чтобы память была выделена по указаному в первом аргументе адресу; без этого флага ядро может выбрать адрес, наиболее близкий к указанному. * `MAP_ANONYMOUS` - выделить страницы в оперативной памяти, а не связать с файлом. * `MAP_SHARED` - выделить страницы, разделяемые с другими процессами; в случае с отображением на файл - синхронизировать изменения так, чтобы они были доступны другим процессам. * `MAP_PRIVATE` - в противоположность `MAP_SHARED`, не делать отображение доступным другим процессам. В случае отображения на файл, он доступен для чтения, а созданные процессом изменения, в файл не сохраняются.

Процессы

В UNIX-системах поддерживается многозадачность, которая обеспечивается: параллельным выполнением нескольких задач, изоляцией адресного пространства каждой задачи.

Полный список процессов можно получить с помощью команды `ps -A`.

Убить какой-то процесс можно с помощью команды `kill`.

Свойства процессов

У каждого процесса существует свои обособленные: * адресное пространство начиная с `0x00000000`; * набор файловых дескрипторов для открытых файлов.

Кроме того, каждый процесс может находиться в одном из состояний: * работает (Running); * приостановлен до возникновения определенного события (Suspended); * приостановлен до явного сигнала о том, что нужно продолжить работу (sTopped); * более не функционирует, не занимает память, но при этом не удален из таблицы процессов (Zombie).

Каждый процесс имеет свой уникальный идентификатор - Process ID (PID), который присваивается системой инкрементально. Множество доступных PID является ограниченным, и его исчерпание проводит к невозможности создания нового процесса (что является механизмом действия форк-бомбы).

Иерархия процессов

Между процессами существуют родственные связи "предок-потомок", таким образом, иерархия процессов представляет собой древовидную структуру. Корнем дерева процессов является процесс с PID=1, который называется `init` (в классических UNIX-системах, в том числе xBSD или консервативных Linux-дистрибутивах), либо `systemd`.

Родительские процессы могут завершаться раньше, чем завершаются их потомки. В этом случае, осиротевшие процессы становятся прямыми потомками процесса с PID=1, то есть `init` или `systemd`.

Процессы можно объединять в *группы процессов* (process group), - множества процессов, которым доставляются *сигналы* о некоторых событиях. Например, в одну группу могут объединяться все процессы, запущенные из одной вкладки приложения-терминала.

Объединение нескольких групп процессов называется *сеансом* (session). Как правило, в сеансы объединяются группы процессов в рамках одного входа пользователя в систему (их может быть несколько, например, несколько входов по `ssh`).

Системный вызов `fork`

Создание нового процесса осуществляется с помощью системного вызова `fork`, который создаёт почти точную копию текущего процесса, причём оба процесса продолжают своё выполнение со следующей, после вызова `fork`, инструкции. Различить родительский процесс от его копии - дочернего процесса, можно по возвращаемому значению `fork`: для родительского процесса возвращается PID вновь созданного процесса, а для дочернего - число 0.

```
pid_t process_id; // в большинстве систем pid_t совпадает с int
if ( -1 == ( process_id=fork() ) ) {
  perror("fork"); // ошибка создания нового процесса
}
else if ( 0 == process_id ) {
  printf("I'm a child process!");
}
else {
  printf("I'm a parent process, created child %d", process_id);
}
```

Ситуация, когда `fork` возвращает значение -1, как правило означает, что система исчерпала допустимый лимит ресурсов на создание новых процессов.

Пример реализации форк-бомбы, назначение которой - исчерпать лимит на количество запущенных процессов:

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

int main()
{
    char * are_you_sure = getenv("ALLOW_FORK_BOMB");
    if (!are_you_sure || 0!=strcmp(are_you_sure, "yes")) {
        fprintf(stderr, "Fork bomb not allowed!\n");
        exit(127);
    }

    pid_t pid;
    do {
        pid = fork();
    } while (-1 != pid);

    printf("Process %d reached out limit on processes\n", getpid());
    while (1) {
        sched_yield();
    }
}
```

Завершение работы процесса

Любой процесс, при добровольном завершении работы, должен явным образом сообщить об этом системе с помощью системного вызова `exit`.

При написании программ на языках Си или C++ этот системный вызов выполняется после завершения работы функции `main`, вызванной из функции `_start`, которая неявным образом генерируется компилятором.

Обратите внимание, что в стандартной библиотеке языка Си уже существует одноименная функция `exit`, поэтому название Си-оболочки для системного вызова начинается с символа подчеркивания: `_exit`.

Функция `exit` отличается от системного вызова `_exit` тем, что предварительно сбрасывает содержимое буферов вывода, а также последовательно вызывает все функции, зарегистрированные с помощью `atexit`.

В качестве аргумента принимается целое число, - код возврата из программы. Несмотря на то, что код возврата является 32-разрядным, к нему применяется операция поразрядного "и" с маской `0xFF`. Таким образом, диапазон кодов возврата находится от 0 до 255.

Код возврата предназначен для того, чтобы сообщить родительскому процессу причину завершения своей работы.

Чтение кода возврата дочернего процесса

Семейство системных вызовов `wait*` предназначено для ожидания завершения работы процесса, и получения информации о том, как процесс жил и умер.

- `wait(int *wstatus)` - ожидание завершения любого дочернего процесса, возвращает информацию о завершении работы;
- `waitpid(pid_t pid, int *wstatus, int options)` - ожидание (возможно неблокирующее) завершения работы конкретного процесса, возвращает информации о завершении работы;
- `wait3(int *wstatus, int options, struct rusage *rusage)` - ожидание (возможно неблокирующее) завершения любого дочернего процесса, возвращает информацию о завершении работы и статистике использования ресурсов;
- `wait4(pid_t pid, int *wstatus, int options, struct rusage *rusage)` - ожидание (возможно неблокирующее) завершения конкретного процесса, возвращает информацию о завершении работы и статистике использования ресурсов.

Если в программе предусмотрено создание более одного дочернего процесса, то использовать системные вызовы `wait` и `wait3` настоятельно не рекомендуется, поскольку дочерние процессы могут завершать свою работу произвольным образом, и это может привести к неоднозначному поведению. Вместо них нужно использовать `waitpid` или `wait4`.

Состояние возврата, которое можно прочитать из таблицы процессов после того, как процесс перестал функционировать, - это причина завершения работы, код возврата, если процесс завершился через `_exit`, или номер убившего его сигнала, если процесс был принудительно завершён. Это состояние закодировано в 32-битном значении, формат которого строго не определен стандартом POSIX. Для извлечения информации используется нобор макросов: * `WIFEXITED(wstatus)` - возвращает значение, отличное от 0, если процесс был завершен с помощью системного вызова `_exit`; * `WIFSIGNALED(wstatus)` - возвращает значение, отличное от 0, если процесс был завершен принудительно; * `WEXITSTATUS(wstatus)` - выделяет код возврата в диапазоне от 0 до 255; * `WTERMSIG(wstatus)` - выделяет номер сигнала, если процесс был завершён принудительно.

Чтение кода возврата, - это не право, а обязанность родительского процесса. В противном случае, дочерний процесс, который завершил свою работу, становится процессом-зомби, информация о завершении которого продолжает занимать место в таблице процессов. Завершение работы родительского процесса при работающих дочерних приводит к тому, что код возврата будет прочитан процессом с PID=1, который автоматически становится родительским для "осиротевших" процессов.

fork-МАГИЯ-exec

Системный вызов exec

Формальное описание системного вызова exec: man 2 execve

Системный вызов `exec` предназначен для замены программы текущего процесса. Как правило, используется совместно с `fork`, но не обязательно.

Си-оболочки для системного вызова `exec` имеют несколько разных сигнатур.

```
int execve(const char *filename,
           char *const argv[],
           char *const envp[]);
int execvpe(.....) // параметры аналогично execve

int execv(const char *filename, char *const argv[])
int execvp(......) // параметры аналогично execv

int execle(const char *filename,
           const char arg0, ..., /* NULL */,
           const char env0, ..., /* NULL */);

int execl(const char *filename,
          const char arg0, ..., /* NULL */);
int execlp(......) // параметры аналогично execl
```

Различные буквы в суффиксах названий `exec` означают? * `v` или `l` - параметры передаются в виде массивов (`v`), заканчивающихся элементом `NULL`, либо в виде переменного количества аргументов (`l`), где признаком конца перечисления аргументов является значение `NULL`. * `e` - кроме аргументов программы передаются переменные окружения в виде строк `КЛЮЧ=ЗНАЧЕНИЕ`. * `p` - именем программы может быть не только имя файла, но и имя, которое нужно найти в одном из каталогов, перечисленных в переменной окружения `PATH`.

Возвращаемым значением `exec` может быть только значение `-1` (признак ошибки). В случае успеха, возвращаемое значение уже в принципе не имеет никакого смысла, поскольку будет выполняться другая программа.

Аргументы программы - это то, что передаётся в функцию `main` (на самом деле, они доступны из `_start`, поскольку располагаются на стеке). Первым аргументом (с индексом `0`), как правило, является имя программы, но это не является обязательным требованием.

Классическим способом запуска новой программы является пара системных вызовов:

```
if (0 == fork) {
  execlp(program, program, NULL);
  perror("exec"); exit(1);
}
```

Замена выполняемой программы с помощью `exec` оставляет неизменным многие аттрибуты процесса, например, открытые файловые дескрипторы, лимиты, и переменные окружения, установленные через `setenv`.

Таким образом, между вызовами `fork` и `exec` можно провести дополнительную настройку программы перед выполнением.

```
if (0 == fork) {
  // Заменить стандартные потоки ввода/вывода на файлы
  // (имитация операции >ВЫХОД <ВХОД в bash)
  close(0);
  close(1);
  /* 0 = */ open(in_file, O_RDONLY);
  /* 1 = */ open(out_file, O_WRONLY|O_CREAT|O_TRUNC, 0640);

  execlp(program, program, NULL);
  perror("exec"); exit(1);
}
```

Для того, чтобы случайно (в достаточно больших программах) не передать открытый файловый дескриптор новой программе, в системном вызове `open` предусмотрен флаг открытия `O_CLOEXEC`, который означает, что файл должен быть закрыт при вызове `exec`.

Лимиты

С процессом связаны некоторые лимиты (ограничения) на используемое процессорное время, максимальный объём памяти, количество файловых дескрипторов, процессов и т. д.

Лимиты подразделяются на *жёсткие*, которые обычные пользователи могут только уменьшать (хотя `root` может и увеличивать), и *мягкие*, которые де-факто являются значениями по умолчанию, и их можно увеличить до жёсткого лимита.

Примерами жёстких лимитов являются ограничения на количество процессов или объём доступной памяти. Пример мягкого лимита - это размер стека, который по умолчанию в Linux равен 8Мб, но может быть изменен произвольным образом, в том числе в большую сторону.

Поскольку изменение размера стека - очень опасная операция, которая может нарушить структуру размещения данных в памяти, изменять этот лимит можно только до запуска функции `main`, либо непосредственно перед выполнением `exec`. В противном случае, поведение программы не определено.

Получение и установка лимитов осуществляются с помощью системных вызовов `getrlimit` и `setrlimit`.

Примеры лимитов см. в get_limits.c.

Пример изменения размера стека см. в shell_with_custom_stack_size.c.

Трассировка выполнения программы

В Linux (не POSIX!) имеется системный вывов `ptrace`, назначение которого - обеспечить возможность отладки.

Первым аргументом `ptrace` является команда, а дальше - некоторое количество аргументов команды.

Если между вызовами `fork` и `exec` выполнить `ptrace(PTRACE_TRACEME,0,0,0)`, то процесс будет приостановлен до тех пор, пока к нему не подключится отладчик, либо программа, которая ведёт себя подобно отладчику, и не разрешит продолжить выполнение дальше.

Некоторые команды, которые программа-"отладчик" может посылать исследуемому процессу:

- `PTRACE_CONT, pid, 0, signo` - продолжить выполнение процесса. Если `signo` не 0, то отправляется сигнал.
- `PTRACE_SINGLESTEP, pid, 0, 0` - выполнить одну инструкцию.
- `PTRACE_SYSCALL, pid, 0, 0` - продолжить выполнение до системного вызова.
- `PTRACE_GETREGS, pid, 0, &state)` - получить значение регистров процессора и сохранить в структуру `user_regs_state`, объявленную в файле `<sys/user.h>`.
- `PTRACE_SETREGS, pid, 0, &state)` - модифицировать регистры процессора.
- `PTRACE_PEEKDATA, pid, addr, 0` - прочитать одно машинное слово из памяти процесса по указанному адресу `addr`.
- `PTRACE_POKEDATA, pid, addr, data` - записать одно машинное слово `data` в память процесса по указанному адресу `addr`.

Пример перехвата системного вызова `write`, который цензурирует нехорошее английское слово в тексте вывода см. в ptrace_catch_string.c.

Сигналы. Часть 1

Введение

Сигнал - это механизм передачи коротких сообщений (номер сигнала), как правило, прерывающий работу процесса, которому он был отправлен.

Сигналы могут быть посланы процессу: * ядром, как правило, в случае критической ошибки выполнения; * другим процессом; * самому себе.

Номера сигналов начинаются с 1. Значение 0 имеет специальное назначение (см. ниже про `kill`). Некоторым номерам сигналов соответствуют стандартные для POSIX названия и назначения, которые подробно описаны `man 7 signal`.

При получении сигнала процесс может: 1. Игнорировать его. Это возможно для всех сигналов, кроме `SIGSTOP` и `SIGKILL`. 2. Обработать отдельной функцией. Кроме `SIGSTOP` и `SIGKILL`. 3. Выполнить действие по умолчанию, предусмотренное назначением стандартного сигнала POSIX. Как правило, это завершение работы процесса.

По умолчанию, все сигналы, кроме `SIGCHILD` (информирование о завершении дочернего процесса) и `SIGURG` (информировании о поступлении TCP-сегмента с приоритетными данными), приводят к завершению работы процесса.

Если процесс был завершён с помощью сигнала, а не с ипользованием системного вызова `exit`, то для него считается не определенным код возврата. Родительский процесс может отследить эту ситуацию, используюя макросы `WIFSIGNALED` и `WTERMSIG`:

```
pid_t child = ...
...
int status;
waitpid(child, &status, 0);
if (WIFEXITED(status)) {
    // дочерний процесс был завершён через exit
    int code = WEXITSTATUS(status); // код возврата
}
if (WIFSIGNALED(status)) {
    // дочерний процесс был завершёл сигналом
    int signum = WTERMSIG(status); // номер сигнала
}
```

Отправить сигнал любому процессу можно с помощью команды `kill`. По умолчанию отправляется сигнал `SIGTERM`, но можно указать в качестве опции, какой именно сигнал нужно отправить. Кроме того, некоторые сигналы отправляются терминалом, например Ctrl+C посылает сигнал `SIGINT`, а Ctrl+\ - сигнал `SIGQUIT`.

Пользовательские сигналы

Изначально в POSIX было зарезервировано два номера сигнала, которые можно было использовать на умотрение пользователя: `SIGUSR1` и `SIGUSR2`.

Кроме того, в Linux предусмотрен диапазон сигналов с номерами от `SIGRTMIN` до `SIGRTMAX`, которые можно использовать на усмотрение пользователя.

Действием по умолчанию для всех "пользовательских" сигналов является завершение работы процесса.

Отправка сигналов программным способом

Системный вызов `kill`

По аналогии с одноимённой командой, `kill` предназначен для отправки сигнала любому процессу.

```
int kill(pid_t pid, int signum); // возвращает 0 или -1, если ошибка
```

Отправлять сигналы можно только тем процессам, которые принадлежат тому пользователю, что и пользователь, по которым выполняется системный вызов `kill`. Исключение составляет пользователь `root`, который может всё. При попытке отправить сигнал процессу другого пользователя, `kill` вернёт значение `-1`.

Номер процесса может быть меньше `1` в случаях: * `0` - отправить сигнал всем процессам текущей группы процессов; * `-1` - отправить сигнал всем процессам пользователя (использовать с осторожностью!); * отрицательное значение `-PID` - отправить сигнал всем процессам группы `PID`.

Номер сигнала может принимать значение `0`, - в этом случае никакой сигнал не будет отправлен, а `kill` вернёт значение `0` в том случае, если процесс (группа) с указанным `pid` существует, и есть права на отправку сигналов.

Функции `raise` и `abort`

Функция `raise` предназначен для отправки сигнала процессом самому себе. Функция стандартной библиотеки `abort` посылает самому себе сигнал `SIGABRT`, и часто используется для генерации исключительных ситуаций, которые получилось диагностировать во время выполнения, например, функцией `assert`.

Системный вызов `alarm`

Системный вызов `alarm` запускает таймер, по истечении которого процесс сам себе отправит сигнал `SIGALRM`.

```
unsigned int alarm(unsigned int seconds);
```

Отменить ранее установленный таймер можно, вызвав `alarm` с параметром `0`. Возвращаемым значением является количество секунд предыдущего установленного таймера.

Обработка сигналов

Сигналы, которые можно перехватить, то есть все, кроме `SIGSTOP` и `SIGKILL`, можно обработать программным способом. Для этого необходимо зарегистрировать функцию-обработчик сигнала.

Системный вызов `signal`

```
#include <signal.h>

// Этот тип определен только в Linux!
typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler); // для Linux
void (*signal(int signum, void (*func)(int))) (int); // по стандарту POSIX
```

Системный вызов `signal` предназначен для того, чтобы зарегистрировать функцию в качестве обработчика определенного сигнала. Первым аргументом является номер сигнала, вторым - указатель на функцию, которая принимает единственный аргумент - номер пришедшего сигнала (т.е. одну функцию можно использовать сразу для нескольких сигналов), и ничего не возвращает.

Два специальных значения функции-обработчика `SIG_DFL` и `SIG_IGN` предназначены для указания обработчика по умолчанию (т.е. отмены ранее зарегистрированного обработчика) и установки игнорирования сигнала.

Системный вызов `signal` возвращает указатель на ранее установленный обработчик.

System-V v.s. BSD

В стандартах, родоначальниками которых были UNIX System-V и BSD UNIX, используется различное поведение обработчика сигнала, зарегистрированного с помощью `signal`. При определении одного из макросов препроцессора: `_BSD_SOURCE`, `_GNU_SOURCE` или `_DEFAULT_SOURCE` (что подразумевается опцией компиляции `-std=gnu99` или `-std=gnu11`), используется семантика BSD; в противном случае (`-std=c99` или `-std=c11`) - семантика System-V.

Отличия BSD от System-V: * В System-V обработчик сигнала выполяется один раз, после чего сбрасывается на обработчик по умолчанию, а в BSD - остается неизменным. * В BSD обработчик сигнала не будет вызван, если в это время уже выполняется обработчик того же самого сигнала, а в System-V это возможно. * В System-V блокирующие системные вызовы (например, `read`) завершают свою работу при поступлении сигнала, а в BSD большинство блокирующих системных вызовов возобновляют свою работу после того, как обработчик сигнала завершает свою работу.

По этой причине, системный вызов `signal` считается устаревшим, и в новом коде использовать его запрещено, за исключением двух ситуаций:

```
signal(signum, SIG_DFL); // сброс на обработчик по умолчанию
signal(signum, SIG_IGN); // игнорирование сигнала
```

Системный вызов `sigaction`

Системный вызов `sigaction`, в отличии от `signal`, в качестве второго аргумента принимает не указатель на функцию, а указатель на структуру `struct sigaction`, с которой, помимо указателя на функцию, хранится дополнительная информация, описывающая семантику обработки сигнала. Поведение обработчиков, зарегистрированных с помощью `sigaction`, не зависит от операционной системы.

```
int sigaction(int signum,
              const struct sigaction *restrict act,
              struct sigaction *oldact);
```

Третьим аргументов является указатель на структуру, описывающую обработчик, который был зарегистрирован для этого. Если эта информация не нужна, то можно передать значение `NULL`.

Основные поля структуры `struct sigaction`: * `sa_handler` - указатель на функцию-обработчик с одним аргументом типа `int`, могут быть использованы значения `SIG_DFL` и `SIG_IGN`; * `sa_flags` - набор флагов, опиывающих поведение обработчика; * `sa_sigaction` - указатель на функцию-обработчик с тремя параметрами, а не одним (используется, если в флагах присутствует `SA_SIGINFO`).

Некоторые флаги, которые можно передавать в `sa_flags`: * `SA_RESTART` - продолжать выполнение прерванных системных вызовов (семантика BSD) после завершения обработки сигнала. По умолчанию (если флаг отсутствует) используется семантика System-V. * `SA_SIGINFO` - вместо функции из `sa_handler` нужно использовать функцию с тремя параметрами `int signum, siginfo_t *info, void *context`, которой помимо номера сигнала, передается дополнительная информация (например PID отправителя) и пользовательский контекст. * `SA_RESETHAND` - после выполнения обработчика

сбросить на обработчик по умолчанию (семантика System-V). По умолчанию (если флаг отсутствует) используется семантика BSD. * `SA_NODEFER` - при повторном приходе сигнала во время выполени обработчика он будет обработан немедленно (семантика System-V). По умолчанию (если флаг отсутствует) используется семантика BSD.

Асинхронность обработки сигналов

Сигнал может прийти процессу в любой момент времени. При этом, выполнение текущего кода будет прервано, и будет запущен обработчик сигнала.

Таким образом, возникает проблема "гонки данных", которая часто встречается в многопоточном программировании.

Существует безопасный целочисленный (32-разрядный) тип данных, для которого гарантируется атомарность чтения/записи при переключении между выполнением основной программы и выполнением обработчика сигнала: `sig_atomic_t`, объявленный в `<signal.h>`.

Кроме того, во время выполнения обработчика сигналов запрещено использовать не потоко-безопасные функции (большинство функций стандартной библиотеки). В то же время, использование системных вызовов - безопасно.

Сигналы. Часть 2

Механизм доставки сигналов

С каждым процессом связан аттрибут, который не наследуется при `fork`, - это *маска сигналов, ожидающих доставки*. Как правило, она представляется внутри системы в виде целого числа, хотя стандартом внутреннее представление не регламентируется. Отдельные биты в этой маске соответствуют отдельным сигналам, которые были отправлены процессу, но ещё не обработаны.

Поскольку одним битом можно закодировать только бинарное значение, то учитывается только сам факт поступления сигнала, но не их количество. Например, это может быть критичным, если сигналы долго не обрабатываются. Таким образом, использовать механизм стандартных сигналов для синхронизации двух процессов - нельзя.

Тот факт, что сигнал оказался в маске ожидающих доставки, ещё не означает, что он будет немедленно обработан. У процесса (или даже у отдельной нити) может существовать маска *заблокированных* сигналов, которая накладывается на маску ожидающих доставки с помощью поразрядной операции `И-НЕ`.

В отличии от маски ожидающих достаки, маска заблокированных сигналов наследуется при `fork`.

Множества сигналов

Множества сигналов описываются типом данных `sigset_t`, объявленным в заголовочном файле `<signal.h>`.

Операции над множествами: * `sigemptyset(sigset_t *set)` - инициализировать пустое множество; * `sigfillset(sigset_t *set)` - инициализировать полное множество; * `sigaddset(sigset_t *set, int signum)` - добавить сигнал к множеству; * `sigdelset(sigset_t *set, int signum)` - убрать сигнал из множества; * `sigismember(sigset_t *set, int signum)` - проверить наличие сигнала в множестве.

Блокировка достаки сигналов

Временная блокировка доставки сигналов часто используется для защиты критических секций программы, когда внезапное выполнение обработчика может повредить целостности данных или нарушению логики поведения.

При этом, нельзя заблокировать сигналы `SIGSTOP` и `SIGKILL`.

Блокировка реализуется установки маски блокируемых сигналов с помощью системного вызова `sigprocmask`:

```
int sigprocmask(int how, sigset_t *set, sigset_t *old_set);
```

где `old_set` - куда записать старую маску (может быть `NULL`, если не интересно), а параметр `how` - это одно из значений: * `SIG_SETMASK` - установить множество сигналов в качестве маски блокируемых сигналов; * `SIG_BLOCK` - добавить множество к маске блокируемых сигналов; * `SIG_UNBLOCK` - убрать множество из маски блокируемых сигналов.

Отложенная обработка сигналов

Сигналы, которые попали в маску сигналов, ожидающих доставки, остаются там до тех пор, пока не будут доставлены (а в дальнейшем - либо игнорированы, либо обработаны). Если сигнал был заблокирован, то его обработчик будет вызван сразу после разблокировки.

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

static void
handler(int signum) {
    static const char Message[] = "Got Ctrl+C\n";
    write(1, Message, sizeof(Message)-1);
}

int main() {
    sigaction(SIGINT,
              &(struct sigaction)
              {.sa_handler=handler, .sa_flags=SA_RESTART},
              NULL);
    sigset_t mask;
    sigemptyset(&mask);
    sigaddset(&mask, SIGINT);

    while (1) {
        sigprocmask(SIG_BLOCK, &mask, NULL);
        sleep(10);
        sigprocmask(SIG_UNBLOCK, &mask, NULL);
    }
}
```

В данном примере sigprocmask.c обработчик сигнала `SIGINT` всё равно будет выполнен, даже несмотря на длительную паузу.

Временная замена маски заблокированных сигналов

Маска сигналов может быть временно заменена.

Системный вызов `sigsuspend`

Системный вызов `sigsuspend(sigset_t *temp_mask)` временно приостанавливает работу программы до тех пор, пока не прийдёт один из сигналов, отсутсвующий в множестве `temp_mask`. Сигналы, отсутсвующие в новом временном множестве, будут доставлены даже в том случае, если они ранее были заблокированы.

Сразу после завершения работы `sigsuspend`, маска заблокированных сигналов вернется в исходную.

Во время обработки сигнала, зарегистрированного `sigaction`

Одно из полей структуры `sigaction` определяет маску сигналов, доставка которых будет заблокирована на время выполнения обработчика. Дополнительные флаги при этом не требуются.

```
struct sigaction act;
memset(&act, 0, sizeof(act));
act.sa_handler = handler;
act.sa_flags = SA_RESTART;
sigfillset(&act.sa_mask); // блокировать все сигналы
```

Сигналы реального времени

Сигналы реального времени - это расширение POSIX, которые, в отличии от стандартных UNIX-сигналов могут быть обработаны используя очередь доставки, и таким образом: * учитывается их количество и порядок прихода; * вместе с сигналом сохраняется дополнительная метаинформация, включая одно челочисленное поле, которое может быть использовано произвольным образом.

Сигналы реального времени задаются значениями от `SIGRTMIN` до `SIGRTMAX`, и могут быть использованы с помощью `kill` как дополнительные стандартные UNIX-сигналы. Действие по умолчанию аналогично `SIGTERM`.

Для использования очереди сигналов, необходимо отправлять их с помощью функции `sigqueue`:

```
#include <signal.h>
union sigval {
    int    sival_int;
    void*  sival_ptr;
};
int sigqueue(pid_t pid, int signum, const union sigval value);
```

Эта функция может завершиться с ошибкой `EAGAIN` в том случае, если исчерпан лимит на количество сигналов в очереди. Опциональное значение, передаваемое в качестве третьего параметра, может быть извлечено получателем из поля `si_value` структуры `siginfo_t`, если использовать вариант обработчика `sigaction` с тремя аргументами.

Дублирование файловых дескрипторов. Каналы

Дублирование файловых дескрипторов

Системный вызов `fcntl` позволяет настраивать различные манипуляции над открытыми файловыми дескрипторами. Одной из команд манипуляции является `F_DUPFD` - создание *копии* дескриптора в текущем процессе, но с другим номером.

Копия подразумевает, что два разных файловых дескриптора связаны с одним открытым файлом в процессе, и разделяют следующие его аттрибуты: * сам файловый объект; * блокировки, связанные с файлом; * текущая позиция файла; * режим открытия (чтение/запись/добавление).

При этом, не сохраняется флаг `CLOEXEC`, который предпиывает автоматическое закрытие файла при выполнении системного вызова `exec`.

Упрощённой семантикой для создания копии файловых дескрипторов являются системные вызовы POSIX: `dup` и `dup2`:

```
#include <unistd.h>

/* Возвращает копию нового файлового дескриптора, при этом, по аналогии
   с open, численное значение нового файлового дескриптора - минимальный
   не занятый номер. */
int dup(int old_fd);

/* Создаёт копию нового файлового дескриптора с явно указанным номером new_fd.
   Если ранее файловый дескриптор new_fd был открыт, то закрывает его. */
int dup2(int old_fd, int new_fd);
```

Неименованные каналы

Канал - это пара связанных между собой файловых дескрипторов, один из которых предназначен для только для чтения, а другой - только для записи.

Канал создается с помощью системного вызова `pipe`:

```
#include <unistd.h>

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

В качестве аргумента системному вызову `pipe` передается указатель на массив и двух целых чисел, куда будут записаны номера файловых дескрипторов: * `pipefd[0]` - файловый дескриптор, предназначенный для чтения; * `pipefd[1]` - файловый дескриптор, предназначенный для записи.

Запись данных в канал

Осуществляется с помощью системного вызова `write`, первым аргументом которого является `pipefd[1]`. Канал является буферизованным, под Linux обычно его размер 65K. Возможные сценарии поведения при записи:

- системный вызов `write` завершается немедленно, если размер данных меньше размера буфера, и в буфере есть место;
- системный вызов `write` приостанавливает выполнение до тех пор, пока не появится место в буфере, то есть предыдущие данные не будут кем-то прочитаны из канала;
- системный вызов `write` завершается с ошибкой `Broken pipe` (доставляется через сигнал `SIGPIPE`), если с противоположной стороны канал был закрыт, и данные читать некому.

Чтение данных из канала

Осуществляется с помощью системного вызова `read`, первым аргументом которого является `pipefd[0]`. Возможные сценарии поведения при чтении:

- если в буфере канала есть данные, то `read` читает их, и завершает свою работу;
- если буфер пустой и есть **хотя бы один** открытый файловый дескриптор с противоположной стороны, то выполнение `read` блокируется;
- если буфер пустой и все файловые дескрипторы с противоположной стороны каналы закрыты, то `read` немедленно завершает работу, возвращая `0`.

Проблема dead lock

При выполнении системных вызовов `fork`, `dup` или `dup2` создаются копии файловых дескрипторов, связанных с каналом. Если не закрывать все лишние (неиспользуемые) копии файловых дескрипторов, предназначенных для записи, то это приводит к тому, что при очередной попытке чтения из канала, `read` вместо того, чтобы завершить работу, будет находиться в ожидании данных.

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

if ( 0!=fork() )  // теперь у нас существует неявная копия файловых дескрипторов
{
    // немного записываем в буфер
    static const char Hello[] = "Hello!";
    write(fds_pair[1], Hello, sizeof(Hello));
    close(fds_pair[1]);

    // а теперь читаем обратно
    char buffer[1024];
    read(fds_pair[0], buffer, sizeof(buffer)); // получаем dead lock!
}
else while (1) shched_yield();
```

Для того, чтобы избежать этой проблемы, необходимо тщательно следить за тем, в какие моменты создаются копии файловых дескрипторов, и закрывать их тогда, когда они не нужны.

Сокеты с установкой соединения

Сокет

Сокет - это файловый дескриптор, открытый как для чтения, так и для записи. Предназначен для взаимодействия: * разных процессов, работающих на одном компьютере (*хосте*); * разных процессов, работающих на разных *хостах*.

Создается сокет с помощью системного вызова `socket`:

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

int socket(
  int domain,    // тип пространства имён
  int type,      // тип взаимодействия через сокет
  int protocol   // номер протокола или 0 для авто-выбора
)
```

Механизм сокетов появился ещё в 80-е годы XX века, когда не было единого стандарта для сетевого взаимодействия, и сокеты являлись абстракцией поверх любого механизма сетевого взаимодействия, поддерживая огромное количество разных протоколов.

В современных системах используемыми можно считать несколько механизмов, определяющих пространство имен сокетов; все остальное - это legacy, которое мы дальше рассматривать не будем.

- `AF_UNIX` (`man 7 unix`) - пространство имен локальных UNIX-сокетов, которые позволяют взаимодействовать разным процессам в пределах одного компьютера, используя в качестве адреса уникальное имя (длиной не более 107 байт) специального файла.
- `AF_INET` (`man 7 ip`) - пространство кортежей, состоящих из 32-битных IPv4 адресов и 16-битных номеров портов. IP-адрес определяет хост, на котором запущен процесс для взаимодействия, а номер порта связан с конкретным процессом на хосте.
- `AF_INET6` (`man 7 ipv6`) - аналогично `AF_INET`, но используется 128-разрядная адресация хостов IPv6; пока этот стандарт поддерживается не всеми хостерами и провайдерами сети Интернет.
- `AF_PACKET` (`man 7 packet`) - взаимодействие на низком уровне.

Через сокеты обычно происходит взаимодействие одним из двух способов (указывается в качестве второго параметра `type`): `* SOCK_STREAM` - взаимодействие с помощью системных вызовов `read` и `write` как с обычным файловым дескриптором. В случае взаимодействия по сети, здесь подразумевается использование протокола `TCP`. `* SOCK_DGRAM` - взаимодейтсвие без предвариательной установки взаимодействия для отправки коротких сообщений. В случае взаимодействия по сети, здесь подразумевается использование протокола `UDP`.

Пара сокетов

Иногда сокеты удобно использовать в качестве механизма взаимодействия между разными потоками или родственными процессами: в отличии от каналов, они являются двусторонними, и кроме того, поддерживают обработку события "закрытие соединения". Пара сокетов создается с помощью системного вызова `socketpair`:

```
int socketpair(
  int domain,    // В Linux поддерживатся только AF_UNIX
  int type,      // SOCK_STREAM или SOCK_DGRAM
  int protocol,  // Только значение 0 в Linux
  int sv[2]      // По аналогии с pipe, массив из двух int
)
```

В отличии от неименованных каналов, которые создаются системным вызовом `pipe`, для пары сокетов не имеет значения, какой элемент массива `sv` использовать для чтения, а какой - для записи, - они являются равноправными.

Использование сокетов в роли клиента

Сокеты могут участвовать во взаимодействии в одной из двух ролей. Процесс может быть *сервером*, то есть объявить некоторый адрес (имя файла, или кортеж из IP-адреса и номера порта) для приема входящих соединений, либо выступать в роли *клиента*, то есть подключиться к какому-то серверу.

Сразу после создания сокета, он ещё не готов к взамиодействию с помощью системных вызовов `read` и `write`. Установка взаимодействия с сервером осуществляется с помощью системного вызова `connect`. После успешного выполнения этого системного вызова - взаимодействие становится возможным до выполнения системного вызова `shutdown`.

```
int connect(
  int sockfd,                  // файловый дескриптор сокета

  const struct sockaddr *addr, // указатель на *абстрактную*
                               // структуру, описывающую
                               // адрес подключения

  socklen_t addrlen            // размер реальной структуры,
                               // которая передается в
                               // качестве второго параметра
)
```

Поскольку язык Си не является объектно-ориентированным, то нужно в качестве адреса передавать: 1. Структуру, первое поле которой содержит целое число со значением, совпадающим с `domain` соответствующего сокета 2. Размер этой структуры.

Конкретными стурктурами, которые "наследуются" от абстрактной структуры `sockaddr` могут быть:

1. Для адресного пространства UNIX - стрктура `sockaddr_un`

```
#include <sys/socket.h>
#include <sys/un.h>

struct sockaddr_un {
  sa_family_t   sun_family;    // нужно записать AF_UNIX
  char          sun_path[108]; // путь к файлу сокета
};
```

2. Для адресации в IPv4 - структура `sockaddr_in` :

```
#include <sys/socket.h>
#include <netinet/in.h>

struct sockaddr_in {
  sa_family_t    sin_family; // нужно записать AF_INET
  in_port_t      sin_port;   // uint16_t номер порта
  struct in_addr sin_addr;   // структура из одного поля:
                             // - in_addr_t s_addr;
                             //   где in_addr_t - это uint32_t
};
```

3. Для адресации в IPv6 - структура `sockaddr_in6` :

```
#include <sys/socket.h>
#include <netinet/in.h>

struct sockaddr_in6 {
  sa_family_t    sin6_family; // нужно записать AF_INET6
  in_port_t      sin6_port;   // uint16_t номер порта
  uint32_t       sin6_flowinfo; // дополнительное поле IPv6
  struct in6_addr sin6_addr;  // структура из одного поля,
                              // объявленного как union {
                              //     uint8_t  [16];
                              //     uint16_t [8];
                              //     uint32_t [4];
                              // };
                              // т.е. размер in6_addr - 128 бит
  uint32_t       sin6_scope_id; // дополнительное поле IPv6
};
```

Адреса в сети IPv4

Адрес хоста в сети IPv4 - это 32-разрядное беззнаковое целое число в *сетевом порядке байт*, то есть Big-Endian. Для номеров портов - аналогично.

Конвертация порядка байт из сетевого в системный и наоборот осуществляется с помощью одной из функций, объявленных в `<arpa/inet.h>` : * `uint32_t htonl(uint32_t hostlong)` - 32-битное из системного в сетевой порядок байт; * `uint32_t ntohl(uint32_t netlong)` - 32-битное из сетевого

в системный порядок байт; * `uint16_t htons(uint16_t hostshort)` - 16-битное из системного в сетевой порядок байт; * `uint16_t ntohs(uint16_t netshort)` - 16-битное из сетевого в системный порядок байт.

IPv4 адреса обычно записывают в десятичной записи, отделяя каждый байт точкой, например: `192.168.1.1`. Такая запись может быть конвертирована из текста в 32-битный адрес с помощью функций `inet_aton` или `inet_addr`.

Закрытие сетевого соединения

Системный вызов `close` предназначен для закрытия *файлового дескриптора*, и его нужно вызывать для того, чтобы освободить запись в таблице файловых дескрипторов. Это является необходимым, но не достаточным требованием при работе с TCP-сокетами.

Помимо закрытия файлового дескриптора, хорошим тоном считается уведомление противоположной стороны о том, что сетевое соединение закрывается

Это уведомление осуществляется с помощью системного вызова `shutdown`.

Использование сокетов в роли сервера

Для использования сокета в роли сервера, необходимо выполнить следующие действия:

1. Связать сокет с некоторым адресом. Для этого используется системный вызов `bind`, параметры которого точно такие же, как для системного вызова `connect`. Если на компьютере более одного IP-адреса, то адрес `0.0.0.0` означает "все адреса". Часто при отладке и возникает проблема, что порт с определенным номером уже был занят на предыдущем запуске программы (и, например, не был корректно закрыт). Это решается принудительным повторным использованием адреса:

```
// В релизной сборке такого обычно быть не должно!
#ifdef DEBUG
int val = 1;
setsockopt(lfd, SOL_SOCKET, SO_REUSEADDR, &val, sizeof(val));
setsockopt(lfd, SOL_SOCKET, SO_REUSEPORT, &val, sizeof(val));
#endif
```

2. Создать очередь, в которой будут находиться входящие, но ещё не принятые подключения. Это делается с помощью системного вызова `listen`, который принимает в качестве параметра максимальное количество ожидающих подключений. Для Linux это значение равно 128, определено в константе `SOMAXCONN`.
3. Принимать по одному соединению с помощью системного вызова `accept`. Второй и третий параметры этого системного вызова могуть быть `NULL`, если нас не интересует адрес того, кто к нам подключился. Системный вызов `accept` блокирует выполнение до тех пор, пока не появится входящее подключение. После чего - возвращает файловый дескриптор нового сокета, который связан с конкретным клиентом, который к нам подключился.

Мультиплексирование ввода-вывода

Неблокирующий ввод-вывод

Системные вызовы `read` и `write` блокируют выполнение текущего потока выполнения в случае отсутствия данных или места в буферах ввода или вывода.

В случае, когда процесс работает одновременно с несколькими файловыми дескрипторами, такое поведение может стать узким местом в производительности.

Для того, чтобы избежать простоя на операциях ввода-вывода, предусмотрен аттрибут файлового дескриптора `O_NONBLOCK`, который можно установить как при открытии файла, так и для уже открытого файлового дескриптора с помощью системного вызова `fcntl`:

```
int fd = ....;                  // какой-то файловый дескриптор
int flags = fcntl(fd, F_GETFL); // получить предыдущие флаги открытия/создания
flags |= O_NONBLOCK;            // добавить флаг неблокируемости
fcntl(fd, F_SETFL, flags);      // установить новые флаги
```

Попытка чтения из файлового дескриптора, если в буфере нет данных, или записи в файловый дескриптор, если в буфере нет места, приведет к ошибке. То есть, системный вызов `read` или `write` завершит работу со значением `-1`, и при этом в переменной `errno` будет зафиксировано значение `EAGAIN`.

В этом случае, можно попробовать выполнить операцию ввода или вывода с файловым дескриптором позже, а тем временем обработать другие файловые дескрипторы.

Обработка событий в Linux

Если основное назначение программы - это обработка данных из файловых дескрипторов, то неблокирующий ввод-вывод может приводить к 100% загрузке процессора даже в том случае, когда с процессом не происходит никакого взаимодействия. Для того, чтобы это избежать, необходимо ставить процесс в состояние ожидания до тех пор, пока не возникнет какое-либо событие, связанное с одним из файловых дескрипторов, требующее реакции.

Такой механизм является системно-зависимым (вне стандарта POSIX); в системе Linux он реализуется через механизм `epoll(7)`, в системах FreeBSD и MacOS - через системный вызов `kqueue`. Эти механизмы реализованы идентично, и различаются только в API.

Очередь ядра

Очередь ядра в системе Linux создается с помощью системного вызова `epoll_create` или `epoll_create1`. Результатом является файловый дескриптор (который должен быть закрыт, когда очередь станет не нужна), который связан с некоторым специальным объектом - очередью ядра, в которую складываются события по некоторому фильтру.

Событие описывается структурой `epoll_event`:

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

typedef union {
    void*       ptr;
    int         fd;
    uint32_t    u32;
    uint64_t    u64;
} epoll_data_t;

struct epoll_event {
    uint32_t        events; // маска событий
    epoll_data_t    data;   // произвольное поле, заполненное при регистрации
};
```

Маска событий - это набор из различных флагов: * `EPOLLIN` - готовность к чтению; * `EPOLLOUT` - готовность к записи; * `EPOLLHUP` - соединение разорвано; * `EPOLLERR` - ошибка ввода-вывода.

Ожидание событий

Перевести процесс в ожидание поступления событий можно с помощью системного вызова `epoll_wait` или `epoll_pwait`.

```
struct epoll_event events[MaxEventsToRead];
int N = epoll_wait(
                   int epoll_fd,  // дескриптор, созданный epoll_create
                   struct epoll_event *events, // куда прочитать события
                   int MaxEventsToRead, // размер массива для чтения
                   int timeout // таймаут в миллисекундах, или -1
                  );
```

Системный вызов `epoll_wait` ожидает появление *хотя бы одного* события из зарегистрированных на наблюдение, блокируя выполнение текущего потока. Поскольку за время простоя процесса или потока может появиться несколько событий, то значение переменной `N` после выполнения `epoll_wait` может быть больше `1`, - это количество событий, которые были прочитаны в массив `events`.

Если был указан параметр `timeout` в значение, отличное от `-1`, либо во время ожидания поступил сигнал, то значение `N` может стать равным `-1`, в `errno` будет записано значение `EINTR`.

Системный вызов `epoll_pwait` дополнительно принимает ещё один аргумент - маску сигналов, которые нужно блокировать на время ожидания.

После выполнения `epoll_wait`, в массив `events` будет записано `N` структур `epoll_event`, которые содержат описание того, что произошло с наблюдаемыми файловыми дескрипторами. При этом, если с одним и тем же файловым дескриптором произошло несколько событий, то они группируются в одно событие.

Регистрация файловых событий для отслеживания изменений

Для того, чтобы события, связанные с определенными файловыми дескрипторами, отслеживались и попадали в очередь ядра, их необходимо явно зарегистрировать с помощью `epoll_ctl`:

```
epoll_ctl(
          int epoll_fd, // дескриптор, созданный epoll_create
          int op, // одна из операций:
                  //  - EPOLL_CTL_ADD - добавить дескриптор в наблюдение
                  //  - EPOLL_CTL_MOD - модицифировать параметры наблюдения
                  //  - EPOLL_CTL_DEL - убрать дескриптор из наблюдения
          int fd, // дескриптор, события над которым нас интересуют
          struct epoll_event *event // структура, в которой описаны:
                                    //  - маска интересуемых событий events
                                    //  - произвольные данные, которые
                                    //    as-is попадают в структуру, читаемую
                                    //    epoll_wait
         );
```

Добавить в один дескприптор `epoll` несколько раз один и тот же файловый дескриптор не получится, - это приведет к ошибке `EEXIST`. Если необходимо модицифировать маску наблюдаемых событий для уже зарегистрированного события, то нужно использовать операцию `EPOLL_CTL_MOD` вместо `EPOLL_CTL_ADD`. В том случае, если всё же необходимо определить разные обработчики событий для одного и того же файлового дескриптора, то можно создать его дубликат с помощью `dup2`.

Если файловый дескриптор был закрыт, то он автоматически удаляется из наблюдаемых файловых дескприпторов.

Отслеживание по изменению фронта (Edge-Triggerd) v.s. отслеживание по состоянию (Level-Triggered)

Описание физической аналогии приведено в статье Edge Triggered v.s. Level Triggered Interrupts (на англ.).

По умолчанию события регистрируются в режиме отслеживания по значению, то есть, по факту наличия флага готовности операций ввода и вывода, если в буфере есть готовые данные или место для записи.

Регистрация обработки событий в режим Eddge-Triggered возможна с указанием флага `EPOLLET`. В этом случае повышается скорость реакции на событие за счет того, что событие регистрируется в тот момент, когда оно только начинает быть готовым, например в буфер ввода начали поступать данные. Недостатком этого подхода является то, что регистрируется только факт изменения состояния, и если, например, не прочитать данные из буфера полностью, то в следующий раз событие готовности к чтению не будет обнаружено, т.к. оно уже произошло.

Событийно-ориентированное программирование

Помимо ввода-вывода, в системе Linux могут быть использованы другие файловые дескрипторы специального назначения, которые также как и обычные, можно регистрировать в наблюдение через `epoll`.

Пара виртуальных сокетов

Пара виртуальных сокетов создается с помощью системного вызова `socketpair`, который, по аналогии с `pipe`, заполняет массив из двух целочисленных значений. Эти файловые дескрипторы могут быть использованы для взаимодействия родственных процессов, от отличаются от неименованных каналов тем, что: 1. Являются двунаправленными 2. Их можно настраивать через `setsockopts`, как обычные сокеты 3. Обрабатывается отдельная операция "завершения соединения", которая, в случае в `epoll` регистрируется как событие `EPOLLIN` с 0 количеством байт.

```
// Пример:
int pair[2];
socketpair(AF_UNIX,     // в Linux поддерживается только UNIX,
           SOCK_STREAM, // еще можно SOCK_DGRAM
           0,           // автоматический выбор протокола
           pair         // массив из 2-х int, куда будут записаны дескрипторы
          );
```

SignalFD, TimerFD, EventFD

Системные вызовы `signalfd`, `timerfd_create`, и `eventfd` реализованы только в Linux, и создают специальные файловые дескрипторы, из которых можно читать события: * поступления определенных сигналов (`signalfd`); * срабатывание таймера (`timerfd_create`); * уведомления, пересылаемые разными потоками (`read` и `write` через `eventfd`).

Многопоточность

Общие сведения

Поток (нить, легковесный процесс) - единица планирования времени в рамках одного процесса.

Все потоки в рамках одного процесса разделяют общее адресное пространство и открытые файловые дескрипторы.

Для каждого потока предусмотрен свой отдельный стек фиксированного размера, который располагается в общем адресном пространстве. В конце стека для каждого потока обычно присутствует небольшой участок памяти (Guad Page), предназначенный для того, чтобы предотвратить ситуацию перезаписи данных другого потока в результате, например, его переполнения.

В каждом процессе существует как минимум один поток, выпонение которого начинается с функции `_start`.

В отличии от обычных процессов, которые имеют иерархическую структуру "родитель-ребенок", все потоки являются равнозначными.

POSIX Threads

Стандартом для UNIX-систем является POSIX Threads API. В системе Linux (как и во FreeBSD), ввиду фрагментации стандартной библиотеки, компоновка программ должна проводиться с опцией компилятора `-pthread`.

В отличии от большинства других функций POSIX, в случае ошибки, функции из pthread не прописывают их код в переменную `errno`, а возвращают различные целочисленные значения, отличные от `0`, которые соответствуют определенным ошибкам.

Создание и запуск нового потока

```
int pthread_create(
    // указатель на переменную-результат
    pthread_t *thread,

    // опционально: параметры нового потока,
    // может быть NULL
    const pthread_attr_t *attr,

    // функция, которая будет выполняться
    (void*)(*function)(void*),

    // аргумент, который передается в функцию
    void *arg
    );
```

Функция `pthread_create` создает новый поток, и сразу же запускает в нем на выполнение функцию, которая передана в качестве аргумента.

Выполняемая функция должна принимать единственный аргумент размером с машинное слово (`void*`), и этот аргумент передается одновременно с созданием потока. Возвращаемое значение выполняемой функции можно будет получить после её выполнения о ожидания завершения потока.

Завершение работы потока и результат работы

Поток завершается в тот момент, когда завершается выполнение функции, либо пока не будет вызван аналог `exit` для потока - функция `pthread_exit`.

Возвращаемые значения размером больше одного машинного слова, которые являются результатом работы потока, не могут быть размещены в стеке, поскольку стек будет уничтожен при завершении работы функции.

Дождаться завершения потока и получить результат можно с помощью функции `pthread_join`

```
int pthread_join(
    // поток, который нужно ждать
    pthread_t thread,

    // указатель на результат работы функции,
    // либо NULL, если он не интересен
    (void*) *retval
    );
```

Функция `pthread_join` ожидает завершения работы определенного потока, и получает результат работы функции.

Возможна ситуация, приводящая к deadlock'у, когда два потока вызывают друг для друга ожидание. Функция `pthread_join` проверяет эту ситуацию, и завершается с ошибкой (не блокируя выполнение).

```
pthread_t a;
pthread_t b;

void* thread_func_a(void *) {
    sleep(1);
    pthread_join(b, NULL);
}

void* thread_func_b(void *) {
    sleep(1);
    pthread_join(a, NULL);
}

// Bug: Deadlock, but detected
pthread_create(&a, NULL, thread_func_a, 0);
pthread_create(&b, NULL, thread_func_b, 0);
```

Такая проверка возможна только при попытке ожидать поток, который уже ожидает поток, пытающийся вызвать `pthread_join`. В случае нескольких потоков, которые косвенно ожидают друг друга, такая диагностика невозможна, и приведет к deadlock'у.

Принудительное завершение потока

Функция `pthread_cancel` принудительно завершает работу потока, если поток явно это не запретил с помощью функции `pthread_setcancelstate`.

```
int pthread_cancel(
    // поток, который нужно прибить
    pthread_t thread
    );
```

Результатом работы функции, который будет передан в `pthread_join` будет специальное значение `PTHREAD_CANCELED`.

В системе Linux остановка потоков реализована через отправку процессом самому себе сигнала реального времени с номером `32`.

Принудительное завершение потока вовсе не означает, что поток будет немедленно остановлен. Функция `pthread_cancel` только проставляет флаг остановки, и этот флаг может быть проверен только во время определенного набора системных вызовов и функций стандартной библиотеки, которые называются *Cancelation Points*.

Полный список функций, которые могут быть прерваны, перечислен в разделе `7` man-страницы `pthreads`.

Некоторые системы, в том числе Linux, позволяют принудительно завершить поток даже вне Cancelation Points. Для этого поток должен вызывать функцию `pthread_setcanceltype` с параметром `PTHREAD_CANCEL_ASYNCHRONOUS`. После этого завершение потока будет осуществляться на уровне планировщика заданий.

Атрибуты потока

Атрибуты потока (второй параметрв в `pthread_create`) хранятся в структуре `pthread_attr_t`, объявление которой является платформо-зависимым, и не регламентируется стандартом POSIX.

Для инициализации атрибутов используется функция `pthread_attr_init(pthread_attr_t *attr)`, и кроме того, после использования, структуру атрибутов необходимо уничтожать с помощью `pthread_attr_destroy`.

С помощью нескольких функций-сеттеров можно задавать определенные параметры вновь создаваемого потока:

- `pthread_attr_setstacksize` - установить размер стека для потока. Размер стека должен быть кратен размеру страницы памяти (обычно 4096 байт), и для него определен минимальный размер, определяемый из параметров системы `sysconf(_SC_THREAD_STACK_MIN)` или константой `PTHREAD_STACK_MIN` из `<limits.h>` (в Linux это 16384 байт);
- `pthread_attr_setstackaddr` - указать явным образом адрес размещения памяти, которая будет использована для стека;
- `pthread_attr_setguardsize` - установить размер защитной области после стека (Guard Page). По умолчанию в Linux этот размер равен размеру страницы памяти, но можно явно указать значение 0.

Синхронизация потоков

Проблема гонки данных

На всех современных процессорных архитектурах операции чтения и записи в память выровненных данных, размер которых не превышает машинного слова, являются атомарными. Однако, с точки зрения строгому следованию стандартам Си и C++, такое предположение неверно, и необходимо использовать специальные типы данных и атомарные операции.

Кроме того, операции над типами данных, размер которых превышает размер машинного слова, заведомо являются неатомарными. В первую очередь, такая проблема проявляется для 64-разрядных типов данных (`double` и `int64_t`) на 32-разрядных архитектурах.

Если несколько потоков или процессов обращаются к одной области памяти, причем один из потоков (процессов) производит запись, то такая ситуация называется *гонкой данных* (data race), и приводит к ситуации неопределенного поведения.

Пример:

```
int64_t balance = 0;

// Thread-1
void* thread_func_one(void *arg) {
    balance += (int64_t) arg;
}

// Thread-2
void* yet_another_thread_func(void *arg) {
    if (balance > 0) {
        // use balance value
    }
    else {
        // fail
    }
}
```

В данной программе гонка данных возникает по причине того, что поток `Thread-2` полагается на неизменность значения переменной `balance` во время работы функции. В то же время, поток `Thread-1` совершенно независимо меняет это значение.

Критические секции

Часть программы, которая подразумевает монопольное использование какого-либо набора переменных, называется *критической секцией*. Начало критической секции подразумевает установки блокировки, которая будет препятствовать выполнению остальных потоков до тех пор, пока блокировка не будет снята.

Стандартным инструментом для обозначения началала критической секции является захват *мьютекса*.

Мьютекс - это примитив синхронизации, который, в большинстве случаев, имеет два состояния. Исключение - рекурсивные мьютексы, которые один поток может захватывать N раз, но остальные потоки не смогут захватить мьютекс до тех пор, пока он не будет N раз освобожден.

Мьютекс объявлен в заголовочном файле `<pthread.h>` , и функции работы с ним требуют линковки с библиотекой `-pthread` .

- `pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr)` - инициализация мьютекса для его последующего использования. Если второй параметр `NULL` , то инициализируется обычный (не рекурсивный) мьютекс. Для создания нового инициализированного мьютекса с параметрами по умолчанию можно использовать макрос `PTHREAD_MUTEX_INITIALIZER` .
- `pthread_mutex_destroy(pthread_mutex_t *mutex)` - уничтожить ранее созданный мьютекс.
- `pthread_mutex_lock(pthread_mutex_t *mutex)` - захватить мьютекс. Если другой поток уже захватил его, то текущий поток приостанавливает свою работу.
- `pthread_mutex_trylock(pthread_mutex_t *mutex)` - пытается захватить мьютекс. В случае успеха возвращает значение `0` , а если мьютекс уже занят, то значение `EBUSY` .
- `pthread_mutex_unlock(pthread_mutex_t *mutex)` - освободить ранее захваченный мьютекс. В отличии от семафоров, освободить мьютекс может только тот поток, которые его захватил, в противном случае это приведет к ошибке `EPERM` .

Условные переменные

Условная переменная - это некоторый примитив синхронизации, используемый для нотификации одним из потоков о наступлении некоторого события, например, готовности данных. Использование условных переменных позволяет отказаться от необходимости применения операций ввода-вывода, и тем самым, исключают необходимость испоьзования системных вызовов.

С условной переменной связан определенный мьютекс, который разблокируется на время ожидания.

- `pthread_cond_init(pthread_cond_t *c, const pthread_condattr_t *attr)` - инициализации условной переменной. Второй параметр может быть `NULL` - в этом случае подразумевается использование переменной только в рамках одного процесса. Для инициализации условной переменной с параметрами по умолчанию используется макрос `PTHREAD_COND_INITIALIZER` .
- `pthread_cond_destroy(pthread_cond_t *c)` - уничтожить условную переменную.
- `pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m)` - ожидает нотификации условной переменной переменной `c` , временно

разблокируя мютекс `m`. Перед вызовом мьютекс должен находиться в заблокированном состоянии, в противном случае - неопределенное поведение. После наступления события нотификации, мьютекс опять блокируется.
- `pthread_cond_timedwait(pthread_cond_t *c, pthread_mutex_t *m, const struct timespec *timeout)` - то же, что и `pthread_cond_wait`, но ожидание прекращается по истечению указанного периода времени.
- `pthread_cond_signal(pthread_cond_t *c)` - уведомляет один поток, для которого выполняется ожидание нотификации. В общем случае, поток выбирается случайным образом, если их несколько.
- `pthread_cond_broadcast(pthread_cond_t *c)` - уведомляет все потоки, для которых выполняется ожидание нотификации.

В случае, если ни один поток не вызвал `pthread_cond_wait`, то нотификация условной переменной проходит незамеченной.

Атомарные переменные и неблокирующие структуры данных

Необходимость блокировки мьютекса, защищающего критическую секцию, может приводить к простою потоков, которые не собираются ничего модифицировать. В ситуации, когда потоков много, это существенно сказывается на производительности, и становится более выгодным использование lock-free структур данных, например односвязных списков.

Основная идея lock-free структур заключается в том, что любая модификация данных проводится каким-либо потоком в области памяти, которая не разделяется с другими потоками. В тот момент времени, когда данные подгтовлены и должны стать доступны остальным потокам, указатель на них записывается в достижимую другим потокам переменную, причем это происходит атомарным образом. Атомарность достигается за счёт того, что размер указателя в памяти не превышает размера машинного слова.

Начиная со 2011 года, в языке Си появилось новое глючевое слово `_Atomic` - модификатор типа, регламентирующий атомарный тип данных. В отличии от языка C++, где атомарным может быть произвольный объект (не совсем честно, поскольку для составных типов используется мьютекс), множество допустимых атомарных типов данных для языка Си ограничивается только типами, которые умещаются в машинное слово. В противном случае, ключевое слово `_Atomic` и атомарные операции не имеют никакого значения. Атомарные операции над типами, объявленными как `_Atomic`, реализуются в Си11 как макросы:

- `void atomic_store(T* object, T value)`,
- `void atomic_store_explicit(T* object, T value, memory_order order)` - сохранить значение в атомарную переменную.
- `T atomic_load(T* object)`,
- `T atomic_load_explicit(T* object, memory_order order)` - загрузить значение из переменной.
- `T atomic_exchange(T* object, T new_value)`,
- `T atomic_exchange_explicit(T* object, T new_value, memory_order order)` - заменить значение и вернуть предыдущее.
- `_Bool atomic_compare_exchange_strong(T* object, T* expected, T new_value)`,
- `_Bool atomic_compare_exchange_strong_explicit(T* object, T* expected, T new_value, memory_order success, memory_order failure)`,
- `_Bool atomic_compare_exchange_weak(T* object, T* expected, T new_value)`,
- `_Bool atomic_compare_exchange_weak_explicit(T* object, T* expected, T new_value, memory_order success, memory_order failure)` - сравнить два значения, в случае их равенства - заменить на новое, в противном случае - записать в `expected` значение `object`.
- `T atomic_fetch_MOD(T* object, T operand)`,
- `T atomic_fetch_MOD_explicit(T* object, T operand, memory_order order)` - получить значение переменной, после чего - модифицировать её. `MOD` можеть быть:
  - `add` - инкремент
  - `sub` - декремент
  - `and` - поразрядное "и"
  - `or` - поразрядное "или"
  - `xor` - поразрядное "исключающее или".

В операции `compare_exchange` вариант `weak` обычно работает быстрее, но может ложно возвращать значение `0` даже при равенстве значений в `object` и `expected`. В некоторых алгоритмах это бывает допустимо, например, если значение проверяется в цикле.

Модели памяти

Компиляторы могут выполнять нетривиальные преобразования программ, применяя различные эвристики о том, как можно увеличить производительность программ. Компилятор имеет полное право выносить некоторые повторяющиеся дейсвтия из тела цикла, хранить значения переменных в регистрах, а не в памяти, и переставлять отдельные операции местами, если это не противоречит семантике исходной *однопоточной* программы.

Все это приводит к тому, что фактически наблюдаемый порядок изменения значений переменных в памяти для одного потока может отличаться от того порядка, которые предполагает другой поток.

Пример:

```
// Thread-1
y = 0;
x = 0;
...
y = 2;
x = 1;

// Thread-2
if (2 == y) {
    z = x;  // 0 или 1?
}
```

В данном случае переменная `z` может иметь значения как `0`, так и `1`, поскольку компилятор мог переставить местами инструкции `y=2` и `x=1`, считая такую перестановку не влияющей на выполнение кода функции, выполняемой в потоке `Thread-1`.

Атомарные операции обычно окружены какими-то соседними инструкциями над обычными (не атомарными) переменными.

Модель памяти (memory order) определяет, какие ограничения накладываются на перестановку обычных операций вокруг атомарных.

Ограничения `memory_order`: * `memory_order_relaxed` - отсутствие каких-либо ограничений на оптимизацию. * `memory_order_consume` - на существующих современных архитектурах совпадает с `memory_order_relaxed`. * `memory_order_acquire` - запрещает перестановку операций работы с памятью вперед данной операции. * `memory_order_release` - запрещает перестановку операций работы с памятью после данной операции. * `memory_order_acq_rel` - одновременно `memory_order_acquire` и `memory_order_release`. * `memory_order_seq_cst` - самые сильные ограничения на перестановку инструкций; все нити видят операции в том порядке, как они были прописаны в коде программы.

По умолчанию (то есть без явного указания модели памяти) подразумевается самая строгая модель `seq_cst`.

Средства межпроцессного взаимодействия POSIX

Разделяемая память

Для создания сегментов разделяемой памяти используется механизм отображаемых файлов mmap. Выполнение системного вызова `mmap` с параметрами `MAP_SHARED` и указанием файлового дескриптора позволяет использовать некоторое имя в файловой системе для взаимодействия между собой неродственных процессов.

Для того, чтобы избежать создания файлов на диске (которые занимают место), предусмотрен механизм создания ортогонального пространства имен ("ключей") для разделяемых файлов, которые существуют только в памяти.

Каждый ключ - это некоторая строка длиной до `NAME_MAX` байт, которая должна начинаться с символа `'/'`. У ключей могут быть права доступа и владелец, по аналогии с обычными файлами.

Такие разделяемые объекты существуют до перезагрузки компьютера, либо до их явного удаления одним из процессов.

Функции для работы с разделяемыми файлами

- `int shm_open(const char *name, int flags, mode_t mode)` - по аналогии с системным вызовом `open`, открывает ключ по имени, как обычный файл. Если среди флагов в `flags` присутствует опция `O_CREAT`, то третий аргумент подразумевает права доступа, в противном случае его значение игнорируется. Возвращает файловый дескриптор, который можно передать в системный вызов `mmap`.
- `int shm_unlink(const char *path)` - удаляет имя из памяти. Если разделяемый файл был имеет отображение в одном или нескольких процессах, то он продолжает быть доступным и занимать место в памяти.

Права доступа можно настраивать с помощью системных вызовов `fchmod` и `fchown`, которые, в отличии от команд shell'а и системных вызовов, работающих с именами, позволяют работать с файловыми дескрипторами.

При создании объекта, он имеет размер `0`, и может быть изменен с помощью `ftruncate`.

Особенности реализации в Linux

В операционной системе Linux (в отличии, например, от FreeBSD), объекты разделяемой памяти - это самые обычные файлы, которые располагаются в файловой системе `tmpfs`, примонтированной в `/dev/shm`.

Кроме того, есть дополнительные особенности: * для использования функций разделяемых объектов POSIX, нужно указывать опцию `-lrt`, посколько `glibc` разбита на несколько частей; * требование про символ `/` в начале имени ключа является не обязательным; тем не менее, это противоречит стандарту `POSIX` и в BSD системах приводит к ошибке `EINVAL`, а в системе `QNX` просто создаст обычный файл на диске в текущем каталоге.

Семафоры

Семафор - это беззнаковая целочисленная переменная, которая обладает дополнительными свойтсвами:

- существуют операции увеличения и уменьшения счетчика, которые выполняются атомарно;
- при попытке уменьшить счетчик, который равен `0`, выполняется приостановка процесса (или нити), который пытается это сделать;
- при увеличении счетчика, если существует какой-то приостановленный процесс, который пытался его уменьшить, работа этого процесса возобновляется.

В теоретической литературе операция *захвата семафора* (уменьшения значения) обычно обозначается буквой `P` (proberen), а операция *освобождения* (увеличение значения) - буквой `V` (verhogen), - названия операций заимствованы из нидерландского языка, т.к. семафоры изобрел Дейкстра.

Семафоры часто используют для синхронизации между собой нескольких потоков выполнения, и в частности, они могут использоваться для предотвращения гонки данных.

Семафоры POSIX

Семафоры POSIX определяются некотроым типом `sem_t`, объявленным в файле `<semaphore.h>`, реализация которого, в общем случае, считается неопределенной, и зависит не только от конкретной операционной системы, но и от процессора.

Функции работы с семафорами обычно принимают его по указателю: * `sem_wait(sem_t *sem)` - захватить семафор (операция `P`); * `sem_post(sem_t *sem)` - освободить семафор (операция `V`); * `sem_trywait(sem_t *sem)` - попытаться захватить семафор, если он равен нулю, то процесс не блокируется, а функция вовзращает значение `-1`, прописывая значение `EAGAIN` в `errno`; * `sem_timedwait(sem_t *sem, struct timespec *timeout)` - захватывает семафор, но если за указанный промежуток времени он не был разблокирован, то функция завершает свою работу с ошибкой `ETIMEDOUT`; * `sem_getvalue(sem_t *sem, int *out_var)` - читает численное значение семафора, не блокируя его; эта функция бывает полезна при отладке.

В системе Linux для использования функций работы с семафорами необходимо компоновать программу с опцией компилятора `-pthread`.

Перед использованием, семафоры должны быть корректно инициализированы. Инициализиция зависит от типа семафора.

Анонимные семафоры

Анонимные семафоры - это семафоры, которые доступны только в рамках одного адресного пространства (для многопоточности), либо родственным процессам.

Создаются с попощью функции `sem_init`:

```
int sem_init(sem_t *sem,    // указатель на семафор в памяти
             int pshared,   // 0 - если предназначен для использования
                            // в рамках одного адресного пространства,
                            // 1 - если разными процессами
             unsinged value // начальное значение
            )
```

Уничтожаются анонимные семафоры с помощью функции `sem_destroy`. При этом ситуация, когда какие-то процессы или нити были заблокированы семафором, считается **неопределенным поведением**, и может приводить к полной блокировке.

Семафоры, которые предназначены для использования разными процессами, должны находиться в разделяемой через `mmap` области памяти, доступной всем задействованным процессам. В противном случае, изменения семафора в одном процессе не будут видны остальным.

Именованные семафоры

Именованные семафоры - это реализация семафоров совместно с механизмом разделяемой памяти POSIX. Имена семафоров подчиняются тем же правилам, что имена сегментов разделяемой памяти, за одним исключением: максимальная длина имени на 4 байта короче, т.к. к имени семафора автоматически приписывается (в зависимости от реализации) суффикс `.sem` или префикс `sem.`.

Создаются именованные семафоры с помощью `sem_open`:

```
sem_t *sem_open(const char *name, int oflag);
sem_t *sem_open(const char *name, int oflag,
                mode_t mode, unsigned int value);
```

По аналогии с `open`, если присутствует флаг `O_CREAT`, то нужно указать права доступа, и кроме того, - начальное значение. В отличии от обычных файлов, не нужно указывать флаги, определяющие режим открытия на чтение/запись. Если открывается существующий семафор, то `oflag=0`.

Закрытие именованного семафора с помощью `sem_close`, в отличии от анонимного, никак не влияет на процессы, которые могут быть им заблокированы: значение счетчика остается неизменным, и может быть изменено после повторного открытия.

Применять операцию `sem_destroy` для именованных семафоров запрещено, так же как и операцию `sem_close` - для анонимных.

Для удаления имени семафора используется функция `sem_unlink(const char *name)`. Как и в случае обычного файла, значение семафора сохраняется даже после удаления до тех пор, пока он не будет закрыт всеми использующими его процессами.

Библиотеки функций и их загрузка

Функции и указатели на них

Код программ в системах с Фон-Неймановской архитектурой размещается в памяти точно так же, как и обычные данные.

Таким образом, он может быть загружен или сгенерирован во время работы программы. Некоторые процессоры позволяют контролировать, какие участки памяти могут быть выполняемые, а какие - нет, и кроме того, это контролируется ядром. Таким образом, выполнить код можно только при условии, что он находится в страницах памяти, помеченных как выполняемые.

Типизация указателей на функции

Объявление вида

```
int (*p_function)(int a, int b);
```

интерптетируется следующим образом: `p_function` - это указатель на функцию, которая принимает два целочисленных аргумента, и возвращает целое знаковое число.

Более общий вид указателя на функцию:

```
typedef ResType (*TypeName)(FuncParameters...);
```

Здесь `ResType` - возвращаемый тип целевой функции, `TypeName` - имя типа-указателя, `FuncParameters...` - параметры функции.

Использование ключевого слова `typedef` является необходимым для языка Си, чтобы каждый раз не писать полностью тип (по аналогии со `struct`).

Объявление указателей на функции необходимо для того, чтобы компилятор знал, как именно использовать адрес какой-то функции, и мог подготовить аргументы, и разбраться с тем, откуда брать возвращаемый результат функции.

Библиотеки

ELF-файл может быть не только исполняемым, но и библиотекой, содержащей функции. Библиотека отличается от исполняемого файла тем, что: * содержит таблицу доступных *символов* - функций и глобальных переменных (можно явно указать её создание опцией `-E`); * может быть размещена произвольным образом, поэтому программа обязана быть скомпилирована в позиционно-независимый код с опцией `-fPIC` или `-fPIE`; * не обязана иметь точку входа в программу - функции `_start` и `main`.

Компиляция библиотеки производится с помощью опции `-shared`:

```
> gcc -fPIC -shared -o libmy_great_library.so lib.c
```

В Linux и xBSD для именования библиотек используется соглашение `libИМЯ.so`, для Mac - `libИМЯ.dynlib`, для Windows - `ИМЯ.dll`.

Связывание программы с библиотекой подразумевает опции: * `-lИМЯ` - указыватся имя библиотеки без префикса `lib` и суффикса `.so`; * `-LПУТЬ` - указывается имя каталога для поиска используемых библиотек.

Runtime Search Path

При загрузке ELF-файла загружаются все необходимые библиотеки, от которых он явно зависит. Посмотреть список зависимостей можно с помощью команды `ldd`.

Библиотеки располагаются в одном из стандартных каталогов: `/lib[64]`, `/usr/lib[64]` или `/usr/local/lib[64]`. Дополнительные каталоги для поиска библиотек определяются в переменной окружения `LD_LIBRARY_PATH`.

Существует возможность явно определить в ELF-файле, где искать необходимые библиотеки. Для этого используется опция линковщика `ld -rpath ПУТЬ`.

Для передачи опций `ld`, который вызывается из `gcc`, используется опция `-Wl,ОПЦИЯ`.

В `rpath` можно указывать как абсолютные пути, так и переменную `$ORIGIN`, которая при загрузке программы раскрывается в каталог, содержащий саму программу. Это позволяет создавать поставку из программы и библиотек, которые не раскиданы по всей файловой системе:

```
> gcc -o program -L. -lmygreat_library program.c \
     -Wl,-rpath -Wl,'$ORIGIN/'.
```

Это создаст выполняемый файл `program`, который использует библиотеку `libmy_great_library.so`, подразумевая, что файл с библиотекой находится в том же каталоге, что и сама программа.

Загрузка библиотек во время выполнения

Библиотеки можно не привязывать намертво к программе, а загружать по мере необходимости. Для этого используется набор функций `dl`, которые

вошли в стандарт POSIX 2001 года.

- `void *dlopen(const char *filename, int flags)` - загружает файл с библиотекой;
- `void *dlsym(void *handle, const char *symbol)` - ищет в библиотеке необходимый символ, и возвращает его адрес;
- `int dlclose(void *handle)` - закрывает библиотеку, и выгружает её из памяти, если она больше в программе не используется;
- `char *dlerror()` - возвращает текст ошибки, связянной с `dl`.

Если `dlopen` или `dlsym` не могут открыть файл или найти символ, то возвращается нулевой указатель.

Пример использования - в файлах lib.c и dynload.c.

Позиционно-независимый исполняемый файл

Опция `-fPIE` компилятора указывает на то, что нужно сгенерировать позиционно-независимый код для `main` и `_start`, а опция `-pie` - о том, что нужно при линковке указать в ELF-файле, что он позиционно-независимый.

Позиционно-независимый выполняемый файл в современных системах размещается по случайному адресу.

Если позиционно-независимый исполняемый файл ещё и содержит таблицу экспортируемых символов, то он одновременно является и библиотекой. Если отсутствует опция `-shared`, то компилятор собирает программу, удаляя из неё таблицу символов. Явным образом сохранение таблицы символов задается опцией `-Wl,-E`.

Пример: ``` # файл abc.c содержит int main() { puts("abc"); }

> gcc -o program -fPIE -pie -Wl,-E abc.c

программа может выполняться как обычная программа

> ./program abc

и может быть использована как библиотека

> python3
>
> > from ctypes import cdll, c_int lib = cdll.LoadLibrary("./program") main = lib["main"] main.restype = c_int ret = main() abc

```
```

Сетевой взаимодействие без установки соединения

Протокол UDP

Схема взаимодействия TCP/IP

После создания сокета типа `SOCK_STREAM`, он должен быть подключен к противоположной стороне с помощью системного вызова `connect`, либо принять входящее подключение с помощью системного выхова `accept`.

После этого становится возможным сетевое взаимодействие с использованием операций ввода-вывода.

Сетевое взаимодействие по TCP/IP (создание сокета с параметрами `AF_INET` и `SOCK_STREAM`) подразумевает, что ядро операционной системы преобразует непрерывный поток данных в последовательность TCP-сегментов, упакованных в IP-пакеты, и наоборот.

Сокеты UDP

Механизм отправки сообщений по UDP подразумевает передачу данных без предварительной установки соединения. Сокет, ориентированный на отправку UDP-сообщений имеет тип `SOCK_DGRAM` и используется совместно с адресацией IPv4 (`AF_INET`) либо IPv6 (`AF_INET6`).

```
// Создание сокета для работы по UDP/IP
int sockfd = socket(AF_INET, SOCK_DGRAM, 0);
```

Как и в случае с TCP, для адресация UDP подразумевает, что помимо IP-адреса хоста необходимо определиться с номером порта, который обслуживает отдельный процесс.

Системные вызовы для передачи и приема данных без установки соединения

```
// Отправить пакет данных
ssize_t sendto(int sockfd,                  // сокет
               const void *buf, size_t len, // данные и размер
               int flags,                   // дополнительные опции
               // адрес назначения (и его размер как для bind/connect)
               const struct sockaddr *dest_addr, socklen_t addrlen);

// Получить пакет данных
ssize_t recvfrom(int sockfd,             // сокет
                 void *buf, size_t len,  // данные и размер
                 int flags,              // дополнительные опции
                 // адрес отправителя (и размер как для accept)
                 const struct sockaddr *src_addr, socklen_t *addrlen);
```

Системный вызов `sendto` предназначен для отправки сообщения. Поскольку предварительно соединение не было установлено, то обязательным является указание адреса назначения: IP-адрес хоста и номер порта.

Системный вызов `recvfrom` предназначен для приема сообщения, и является блокирующим до тех пор, пока придет хотя бы одно сообщение UDP.

Размер буфера, в который `recvfrom` должен записать данные, должен быть достаточного размера для хранения сообщения, в противном случае данные, которые не влезли в буфер, будут потеряны.

Для того, чтобы иметь возможность принимать данные по UDP, необходимо анонсировать прослушивание определенного порта с помощью системного вызова `bind`; параметры адреса для `recvfrom` предназначены только для получения информации об отправителе, и являются опциональными (эти значения могут быть NULL).

Инструменты в Linux для отладки сетевого взаимодействия

Сетевой ввод-вывод

Команда `nc` (сокращение от `netcat`) работает аналогично команде `cat`, но в качестве аргумента принимает не имя файла для вывода потока данных, а пару `хост порт`. Параметр `-u` означает отправку UDP-пакета.

Если предполагается использовать только адресацию IPv4, но не IPv6, то используется опция `-4`.

```
# Пример: передать данные из in.dat в UDP-сокет на localhost
# порт 3000 и записать вывод в файл out.dat
> cat in.dat | nc -4 -u localhost 3000 >out.dat
```

Режим Бога

Утилита `wireshark` позволяют просматривать абсолютно все пакеты на уровне от `Ethernet`, которые проходят через систему. Для этого требуются права `root`, либо настройка `Linux Capabilities` для команды `/usr/bin/dumpcap`, которая является частью `wireshark`:

```
sudo setcap cap_net_raw,cap_net_admin+eip /usr/bin/dumpcap
```

Поскольку через систему проходит много сетевых пакетов, то для поиска только интересующих пакетов необходимо настроить фильтр.

Python

Стандартная библиотека Python содержит средства работы с сокетами, которые в точности соответствуют их аналогам для POSIX.

Пример отправки UDP-сообщения:

```
from socket import socket, AF_INET, SOCK_DGRAM

IP = "127.0.0.1"
PORT = 3000

sock = socket(AF_INET, SOCK_DGRAM)   # создание UDP-сокета
# Соединение не требуется
sock.sendto("Hello!\n", (IP, PORT))  # отправка сообщения
```

Прием UDP-сообщений:

```
from socket import socket, AF_INET, SOCK_DGRAM

IP = "127.0.0.1"
PORT = 3000
MAX_SIZE = 1024

sock = socket(AF_INET, SOCK_DGRAM)       # создание UDP-сокета
sock.bind((IP, PORT))                    # нужно анонсировать порт

while True:
    data, addr = sock.recvfrom(MAX_SIZE) # получить сообщение
    print("Got {} from {}", data, addr)
```

Linux Capabilities

В классических UNIX-системах права процессов на выполнение привилегированных действий разграничиваются только на уровне доступа к файлам, либо на уровне "обычный пользователь" - "администратор". В современных ядрах Linux существует ортогональный механизм для предоставления отдельным программам определенных прав, не связанных с доступом к файлам, который называется capabilities(7).

Отдельному исполняемому файлу можно назначить маску привилегированных разрешений, которые распространяются только на отдельную программу (но не дочерние процессы) с помощью утилиты `setcap` (требуются права root для запуска).

Формат вызова:

```
> sudo setcap CAPABILITIES+FLAGS EXECUTABLE_FILE
```

Здесь `CAPABILITIES` - одно, либо несколько, разделенных запятыми, полномочий. `FLAGS` - это комбинация флагов:

- `p` - (Permitted) - полномочие разрешено для исполняемого файла;
- `i` - (Inherited) - может наследоваться при вызове `exec`, но это не распространяется при создании дочернего процесса через `fork`;
- `e` - (Effective) - набор полномочий добавляется в существующее множество разрешений, а не заменяет его.

При этом, установленные атрибуты `capabilities` не сохраняются:

- во время модификации файла (например, в результате перекомпиляции);
- при копировании или переименовании файла.

Таким образом, чтобы иметь возможность создавать и отлаживать программу, требующую дополнительные полномочия, необходимо обеспечить вызов `setcap` на этапе установки или сборки.

Так как `capabilities` это атрибуты файлов, то для их работы требуется поддержка со стороны файловой системы. В стандартных для linux файловых системах с этим проблем нет, но если файл находится на примонтированном разделе с неподдерживающей `capabilities` файловой системой, то попытка установки закончится ошибкой `Failed to set capabilities on file './executable' (Operation not supported)`.

Также бывает удобным (для отладки) поставить необходимый набор полномочий на отладчик `gdb`; для корректной работы это требует дополнительно установки того же набора полномочий на командный интерпретатор `bash`.

Взаимодействие на уровне сетевого интерфейса

Пакетные сокеты

Система Linux позволяет взаимодействовать с сетевыми устройствами на низком уровне, используя специальный тип сокетов: пакетные сокеты `AF_PACKET`.

Более подробно работа с сокетами на низком уровне рассмотрена в статье Introduction to RAW Sockets

Для создания таких сокетов требуются либо права `root`, либо настройка `cap_net_raw`, в противном случае системный вызов `socket` вернет значение `-1`.

При работе с обычными TCP или UDP сокетами, ядро операционной системы полностью абстрагирует пользовательский процесс от дополнительной информации, связанной с доставкой сетевых данных.

При работе с пакетными сокетами необходимо самостоятельно реализовывать обработку требуемых заголовков.

Существует два уровня абстракции для пакетных сокетов: передача данных, которые заворачиваются в стандартный фрейм Ethernet `(AF_PACKET, SOCK_DGRAM)`, там и полностью произвольный поток данных `(AF_PACKET, SOCK_RAW)`, который может быть использован, например, для отправки широковещательных Ethernet-фреймов.

Бинарные заголовки сетевых протоколов

Для работы с заголовками сетевых протоколов средствами языков Си/C++ можно использовать обычные структуры.

Порядок, в котором объявлены поля структуры в тексте программы, является при этом существенным, поскольку он соответствует тому, в каком порядке хранятся данные. Кроме того, необходимо учитывать тот факт, что компиляторы оптимизируют код, выравнивания поля структур в соответствии с особенностями архитектур процессоров, и необходимо явным образом указывать использование "упакованных" структур.

Пример: заголовок Ethernet-кадра может быть представлен следующим образом.

```
typedef struct {
    /* MAC-адрес получателя, 6 байт */
    uint8_t destination[6];
    /* MAC-адрес отправителя, 6 байт */
    uint8_t source[6];
    /* Тип передаваемого пакета */
    uint16_t type;
} __attribute__((__packed__)) ethernet_header_t;
```

Кроме того, необходимо помнить о том, что большинство сетевых протоколов подразумевают использование сетевого порядка байт, поэтому нужно использовать функции `htons`, `ntohs`, и др., для того, чтобы правильно представлять целочисленные значения.

Адресация без IP-адреса

У каждого сетевого интерфейса есть имя в системе, например `eth0` или `wlan0`, которое можно посмотреть в выводе команды `ifconfig`, и *порядковый номер* (индекс). У каждого, даже не настроенного, сетевого интерфейса есть свой аппаратный адрес (MAC-адрес), размер которого обычно 6 байт.

При адресации через семейство протоколов `AF_PACKET` используется структура `sockaddr_ll`:

```
struct sockaddr_ll {
    unsigned short sll_family;   /* Always AF_PACKET */
    unsigned short sll_protocol; /* Physical-layer protocol */
    int            sll_ifindex;  /* Interface number */
    unsigned short sll_hatype;   /* ARP hardware type */
    unsigned char  sll_pkttype;  /* Packet type */
    unsigned char  sll_halen;    /* Length of address */
    unsigned char  sll_addr[8];  /* Physical-layer address */
};
```

Поле `sll_family` должно иметь значение `AF_PACKET` (поскольку необходимо отделять этот тип адресов от других возможных `struct sockaddr`).

Для отправки низкоуровневых пакетов определенному устройству с использованием протокола Ethernet, когда используется комбинация `(AF_PACKET, SOCK_DGRAM)`, необходимо заполнять поля:

- `sll_protocol` - значение константы из `<linux/if_ether.h>`, которая определят тип пакета данных (протокол), который содержится внутри Ethernet-фрейма;
- `sll_halen` - длина адреса в байтах; для современных реализаций Ethernet это значение равно `6` (константа `ETH_ALEN` из `<linux/if_ether.h>`);
- `sll_ifindex` - индекс сетевого устройства; нумерация начинается с `1`, специальное значение `0` может быть использовано только для чтения (признак того, что интересуют данные из любого устройства);
- `sll_addr` - значение MAC-адреса.
- Все остальные поля заполняются драйвером устройства и должны быть инициализированы нулями.

Если используется отправка пакетов без заголовка Ethernet, то есть, используется комбинация `(AF_PACKET, SOCK_RAW)`, то достаточно указать только порядковый индекс сетевого интерфейса `sll_ifindex`.

Управление устройствами ввода-вывода

Для управления файло-подобными устройствами ввода-вывода используется системный вызов `ioctl`, сигнатура которого такая же, как для `fcntl`: первый аргумент - это файловый дескриптор, затем целочисленная команда, а потом возможны аргументы произвольного типа, в зависимости от команды.

Набор команд для работы с сетевыми интерфейсами описан в man 7 netdevice. Многие из них могут быть выполнены только при наличии соответствующих прав (если модифицируют параметры сетевого интерфейса). С помощью GET-команд, отправляемых через системный вызов `ioctl`, можно выяснить индекс устройства по его имени, связанный с ним MAC-адрес, IP-адрес, если устройство настроено, и т. д.

Berkley Packet Filter

Основной класс задач, в которых применяется чтение из RAW-сокетов, - это мониторинг системы. При этом возникает проблема большого потока данных, который необходимо обрабатывать, и постоянное переключение контекста между выполнением кода в пространстве ядра и в пространстве пользователя существенно снижает производительность.

Поскольку принимать нужно не все пакеты, проходящие через сетевой интерфейс, а только те, которые соответствуют некоторым критериям, то логично перенести логику фильтрации в адресное пространство ядра, а затем получать от ядра только те пакеты, которые не отвергнуты фильтром.

В качестве примера использования можно рассмотреть утилиту `tcpdump`, которая принимает в качестве аргумента текстовую строку, описывающую функцию фильтрации, и отображает только те события, которые соответствуют фильтру. В своей реализации утилита `tcpdump` использует BPF.

Дополнительные материалы (English only):

- Документация из поставки ядра Linux: Linux Socket Filtering aka Berkley Packet Filter (BPF)
- man 2 bpf
- BPF and XDP Reference Guide

Классический BPF: Linux Socket Filter

Мониторинг сети и задача фильтрации

Рассмотрим задачу фильрации пакетов на уровне Data Link Layer, и будем отлавливать те из них, которые соответствуют некоторому критерию. Для простоты можно рассмотреть IPv4/UDP-сообщения к определенному DNS-серверу, - такие запросы будет легко отлаживать.

Создадим Data-Link сокет, и свяжем его с определенным сетевым интерфейсом, например `eth0`:

```
int sock = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL));

/* Не забываем проверять ошибки! Packet-сокет невозможно создать,
не имея права root или настроенный CAP_NET_RAW */
if (-1==sock) { perror("socket"); exit(1); }

/* Далее нужно связать сокет с определенным сетевым интерфейсом */
struct ifreq req;
memset(&req, 0, sizeof(req));
strncpy(req.ifr_name, "eth0", IFNAMSIZ);
ioctl(sock, SIOCGIFINDEX, &req, sizeof(req)); // определяем индекс eth0

/* Для Data-Link Layer нужно заполнить только часть полей адреса,
остальные должны быть инициализированы нулями */
struct sockaddr_ll addr;
memset(&addr, 0, sizeof(addr));
addr.sll_family = AF_PACKET;  // указываем, что структура для PACKET
addr.sll_protocol = htons(ETH_P_ALL);  // нас интересует только Ethernet
addr.sll_ifindex = req.ifr_ifindex; // индекс устройства (см. выше)

/* Привязываем сокет к определенному адресу. Если не вызывать bind,
то нужно использовать recvfrom и sendto с явным указанием адреса */
if (-1==bind(sock, (struct sockaddr*)&addr, sizeof(addr))) {
    perror("bind"); exit(1);
}
```

Теперь можно наблюдать за тем, что проходит через этот сетевой интерфейс.

```
for (;;) {
    char buffer[4096];
    memset(buffer, 0, sizeof(buffer));

    /* Читаем блок данных из сетевого устройства */
    size_t cnt = recv(sock, buffer, sizeof(buffer), 0);

    uint32_t from_ip, to_ip;

    /* Извлекаем адреса источника и получателя из заголовка IPv4 */
    memcpy(&from_ip, buffer+26, sizeof(from_ip));
    memcpy(&to_ip, buffer+30, sizeof(to_ip));

    char from_addr[20], to_addr[20];
    memset(from_addr, 0, sizeof(from_addr));
    memset(to_addr, 0, sizeof(to_addr));
    inet_ntop(AF_INET, &from_ip, from_addr, sizeof(from_addr));
    inet_ntop(AF_INET, &to_ip, to_addr, sizeof(to_addr));

    printf("Got communication from %s to %s\n", from_addr, to_addr);
}
```

На реально используемой системе можно будет наблюдать огромное количество пакетов сетевого взаимодействия.

Виртуальная машина Classic-BPF

Программа фильтрации, загружаемая в ядро, состоит из набора 64-битных RISC-команд, которые выполняются виртуальной машиной, либо могут быть транслированы в нативный код.

Каждая инструкция кодируется следующим образом:

```
struct sock_filter {
    __u16   code;   // 16 бит - код команды
    __u8    jt;     // 8 бит - смещение для true/jump-инструкций
    __u8    jf;     // 8 бит - смещение для false/jump-инструкций
    __u32   k;      // 32 бит - поле для произвольных данных
};
```

У виртуальной машины есть только два 32-битных регистра: аккумулятор `A`, над которым можно выолнять произвольные действия, и счетчик инструкций `X`. Возможен доступ к "памяти", при этом адресуется содержимое исследуемого сетевого пакета.

Поскольку виртуальная машина была спроектирована по аналогии с реально существующим процессором Motorola 6502, то для этого набора команд существует язык ассемблера. Программная реализация ассемблера BPF находится в поставке исходных текстах ядра Linux: `tools/bpf/bpf_asm`.

Команды ассемблера BPF

- Перемещение данных:
  - загрузить в регистр `A`: `ld` - слово, `ldh` - полуслово, `ldb` - байт;
  - сохранить значение в памяти: `st` для регистра `A`, `stx` для регистра `X`;
  - перемещение между регистрами: `tax` - из `A` в `X`, `txa` - из `X` в `A`
- Арифметические операции над регистром `A`: `add`, `sub`, `mul`, `div`, `mod`, `neg`, `and`, `or`, `xor`, `lsh`, `rsh`
- Переходы на метку:
  - `jmp` - безусловный переход;
  - `jeq`, `jne`, `jlt`, `jle`, `jgt`, `jge` - условный переход, при этом можно опционально указать вторую метку, на которую будет выполнен переход в случае не выполнения условия
- Завершение работы: команда `ret` завершает работу и возвращает результат обработки фильтра.

Пример программы на cBPF

Рассмотрим задачу фильтрации Ethernet-фреймов: будем принимать только фреймы, внутри которых содержатся UDP-сообщения, адресованные DNS Google по адресу `8.8.8.8`.

```
filter_google_dns:
    ldh     [12]               ; 16-бит значение после двух MAC-адресов
    jne     #0x0800, fail      ; проверяем, что внутри кадра у нас IPv4-пакет
    ldb     [23]               ; 8-бит значение типа протокола в заголовке IP
    jne     #17, fail          ; 17 - это UDP, 6 - это TCP
    ld      [30]               ; 4-байтное значение IP-адреса
    jne     #0x08080808, fail  ; сравниваем с адресом 8.8.8.8
success:
    ret     #-1                ; значение -1 == 0xFFFFFFFF
fail:
    ret     #0                 ; значение  0
```

Данная программа проверяет, что внутри Ethernet-фрейма содержится действительно IPv4-пакет, который, в свою очередь, содержит сообщение типа UDP, и адресован получателю `8.8.8.8`. Возвращаемое значение - это максимальное количество байт, которое фильтр должен пропустить. Таким образом, значение `0` означает отклонение пакета, а максимально возможное беззнаковое целочисленное значение - пропуск пакета целиком.

Загрузка программы-фильтра в ядро

К сокету можно прикрепить BPF-фильтр, используя системный вызов `setsockopt`:

```
setsockopt(
    sock,               // файловый дескриптор сокета
    SOL_SOCKET,         // опция предназначена для сокета в целом
    SO_ATTACH_FILTER,   // команда "присоединить фильтр"

    // указатель на структуру, которая содержит cBPF-программу
    struct *sock_fprog program,
    // размер аргумента; требуется как generic-параметр для setsockopt
    sizeof(struct sock_fprog)
);
```

Сама структура программы состоит из двух полей: указателя на последовательность инструкций, и количество инструкций (не байт!) в программе. Каждая инструкция - это 8 байт, которые можно описать структурой `struct sock_filter`.

Ассемблер BPF, который имеется в составе исходных текстов ядра Linux, имеет опцию для вывода байткода в формате Си-структур, и этот вывод можно включить в код препроцессором.

```
> linux-5.15/tools/bpf/bpf_asm -c filter.s >filter.inc
```

```
struct sock_filter[] code = {
    #incldue "filter.inc"
    /*
    Здесь препроцессор вставит как есть текст вывода ассемблера:
    { 0x28,  0,  0, 0x0000000c },
    { 0x15,  0,  5, 0x00000800 },
    { 0x30,  0,  0, 0x00000017 },
    { 0x15,  0,  3, 0x00000011 },
    { 0x20,  0,  0, 0x0000001e },
    { 0x15,  0,  1, 0x08080808 },
    { 0x06,  0,  0, 0xffffffff },
    { 0x06,  0,  0, 0000000000 },
    */
};

struct sock_fprog program = {
    .filter = code,   // указатель на последовательность инструкций

    /* Количество инструкций можно рассчитать как размер всех
    инструкций в байтах, деленный на размер одной инструкции */
    .len = sizeof(code)/sizeof(code[0])
};
```

Загрузка программы подразумевает её обязательную проверку верификатором, который проверяет, что: 1) все инструкции в программе корректны; 2) размер программы не превышает 4096 инструкций; 3) программа не содержит циклов.

В случае отклонения программы верификатором, системный вызов `setsockopt` вернёт значение `-1`.

Linux Extended BPF

Начиная с ядра 3.19 в ядре Linux появилась новая реализация BPF, которая реализует виртуальную машину со следующими свойствами:

- 11 регистров вместо 2, 10 из них - общего назначения
- все регистры - 64-битные
- появился стек, таким образом можно делать вложенные функции
- вызовы некоторых встроенных функций в адресном пространстве ядра.

Начиная с версии ядра 4.1, кроме возможности фильтрации сетевых пакетов, виртуальная машина EBPF позволяет выполнять код при наступлении определенных событий ядра.

Кроме того, EBPF-программы могут использовать постоянные хранилища данных (`maps`), которые доступны также из адресного пространства процесса: массивы и хеш-таблицы.

Таким образом, область применения расширилась до трассировки и измерения производительности.

Системный вызов bpf

```
int bpf(
    // Команда взаимодействия с подсистемой EBPF
    int cmd,
    // Аргумент команды
    union bpf_attr *attr,
    // Размер аргумента команды
    unsigned int size
);
```

Использование системного вызова `bpf` на момент 02 апреля 2020 подразумевает использование функции `syscall` , поскольку Си-оболочка для него не реализована в `glibc` . Кроме того, многие возможности пока ещё не задокументированы.

Основные команды для `bpf` :

- `BPF_PROG_LOAD` - загрузить программу в ядро, и проверить её верификатором; возвращает дескриптор программы
- `BPF_MAP_CREATE` , и другие команды `BPF_MAP_*` - создание постоянного хранилища, и операции над ним.

Для каждой команды существует отдельная структура аргумента, описанная в `<linux/bpf.h>` .

Программы, в свою очередь, могут быть разными по назначению:

- `BPF_PROG_TYPE_SOCKET_FILTER` - простой фильтр, как в Classic BPF
- `BPF_PROG_TYPE_KPROBE` - программа для обработки событий ядра `kprobe`
- `BPF_PROG_TYPE_XDP` - продвинутая фильтрация пакетов как в файрволе
- [и ещё много типов - см `<uapi/linux/bpf.h>`]

Программа должна содержать функцию - точку входа, единственным аргументом которой, в регистре `R1` , будет указатель на контекст выполнения, тип которого зависит от типа программы.

Загрузка программы

Программа (возможно) загружается в ядро системы после вызова `bpf` :

```
/* Приходится использовать syscall, т.к. нет оболочки в glibc */
int program_id = syscall(
    SYS_bpf,               // номер системного вызова bpf
    BPF_PROG_LOAD,         // команда для загрузки программы
    &bpf_argument,         // указатель на аргумент команды
    sizeof(bpf_argument)   // ... и размер аргумента
);

/* В случае успеха, будет возвращен файловый дескриптор программы.
   Для освобождения ресурсов, когда программа станет не нужна,
   нужно использовать обычный close() */

close(program_id);
```

Аргумент команды для загрузки - это структура (точнее, объявленная как `union`), в которой должны быть заполнены поля:

```
 char bpf_code[4096*8] = ....;
char loader_log[65535];

union bpf_attr bpf_argument = {
    .prog_type = BPF_PROG_TYPE_SOCKET_FILTER,  // тип программы
    .insns = bpf_code,                         // указатель на бинарный код
    .insn_cnt = sizeof(bpf_code) / 8,          // количество инструкций
    .log_level = 1,                            // вести ли лог загрузки?
    .log_buf = loader_log,                     // куда писать лог загрузки
    .log_size = sizeof(loader_log),            // размер буфера для лога
    .license = "GPL"                           // должна быть GPL-совместимая
};
```

Лог загрузки содержит текст, который очень полезен при отладке, поскольку не любая программа будет считаться корректной с точки зрения верификатора.

Загруженную программу, если это фильтр для сокета, можно прикрутить к сокету:

```
setsockopt(sock, SOL_SOCKET, SO_ATTACH_BPF, &program_id, sizeof(program_id));
```

Байт-код EBPF-программ

Формат EBPF программ отличается от Classic BPF, поэтому использовать старый ассемблер не получится.

Формат инструкции:

```
struct bpf_insn {
    __u8  code,        // 8-бит код команды
    __u8  dst_reg:4,   // 4-бит регистр-назначение
    __u8  src_ref:4,   // 4-бит регистр-источник
    __s16 off,         // 16-бит значение для относительного адреса
    __s32 imm          // 32-бит значение для кодирования констант
};
```

Компиляция EBPF-программ

Компилировать EBPF-программы можно с помощью свежих версий тулкита CLang/LLVM. Необходимо проверить, что LLVM поддерживает цель `bpf`:

```
> llc --version
LLVM (http://llvm.org/):
  LLVM version 9.0.1
  Optimized build.
  Default target: x86_64-unknown-linux-gnu
  Host CPU: skylake

  Registered Targets:
     ......
    bpf        - BPF (host endian)
     ......
```

Рассмотрим компиляцию тривиальной программы, которая запрещает все пакеты:

```
/* trivial.c */
int trivial_socket_filter(void *ctx) {
    return 0;  // change to -1 to allow all
}
```

Скомпилируем эту программу в объектный файл для цели `bpf`:

```
> clang -c -target bpf trivial.c
```

В результате получим объектный файл, который содержит примерно такой код:

```
> llvm-objdump -d trivial.o
trivial.o:      file format ELF64-BPF

Disassembly of section .text:

0000000000000000 trivial_socket_filter:
       0:       7b 1a f8 ff 00 00 00 00 *(u64 *)(r10 - 8) = r1
       1:       b7 00 00 00 00 00 00 00 r0 = 0
       2:       95 00 00 00 00 00 00 00 exit
```

Нулевая инструкция загружает в стек переменную `ctx`, первая присваивает значение `r0 = 0` (по соглашению о вызовах, это возвращаемое значение), а вторая выполняет выход из функции.

Для того, извлечь из объектного файла кусок кода, можно использовать утилиту `objcopy`. В дальнейшем этот код можно будет загрузить в программу как простые бинарные данные.

```
> llvm-objcopy     \    # требуется использовать objcopy из поставки LLVM
         -O binary \    # выходной формат - бинарный
         -j .text  \    # копируем только секцию .text
         trivial.o \    # имя входного файла
         trivial.bin    # имя файла с результатом
```

С помощью `objcopy` можно получить бинарный файл размером 24 байта, который содержит только код (ровно три инструкции), но не заголовки и таблицы.

```
> hexdump -C trivial.bin
00000000  7b 1a f8 ff 00 00 00 00  b7 00 00 00 00 00 00 00  |{...............|
00000010  95 00 00 00 00 00 00 00                           |........|
00000018
```

Ограничения на EBPF-программы

Как и в случае с Classic BPF, загружаемые в ядро программы проходят строгую валидацию. Размер программы может составлять 4096 инструкций (до Linux 5.1), либо до миллиона инструкций (начиная с версии Linux 5.1).

Так же валидатор проверяет, что программа гарантированно завершится за конечное время, поэтому использовать циклы в Си-программах можно только если на этапе компиляции известно число итераций. Для того, чтобы компилятор "развернул" циклы в длинную линейную программу, нужно перед циклом указать директиву `#pragma unroll` и указать опцию компиляции `-funroll-loops`.

Функции стандартной Си-библиотеки становятся не доступны, поскольку программа выполняется в ограниченном окружении. Тем не менее, есть набор функций, доступных для EBPF-программ, они перечислены в man 7 bpf-helpers.

Ещё одним ограничением является порядок доступа к данным в памяти. Например, он должен быть выровненным, и вполне безобидная конструкция не пройдёт проверку валидатором:

```
ctx = 0;   // начало пакета
unsigned int ip_dest = *(unsigned int*)(ctx+30); // извлекаем IP-адрес
```

Вывод лога валидатора:

```
6: (61) r1 = *(u32 *)(r1 +30)
invalid bpf_context access off=30 size=4
```

Поэтому приходится использовать низкоуровневые LLVM-функции для доступа к данным напрямую:

```
/* llvm builtin functions that eBPF C program may use to
 * emit BPF_LD_ABS and BPF_LD_IND instructions
 */
unsigned long long load_byte(void *skb,
                             unsigned long long off) asm("llvm.bpf.load.byte");
unsigned long long load_half(void *skb,
                             unsigned long long off) asm("llvm.bpf.load.half");
unsigned long long load_word(void *skb,
                             unsigned long long off) asm("llvm.bpf.load.word");
```

BPF Compiler Collection

See tutorial: bcc Python Developer Tutorial

Протокол HTTP и библиотека cURL

Протокол HTTP

Общие сведения

Протокол HTTP используется преимущественно браузерами для загрузки и отправки контента. Кроме того, благодаря своей простоте и универсальности, он часто используется как высокоуровневый протокол клиент-серверного взаимодействия.

Большинтсво серверов работают с версией протокола `HTTP/1.1`, который подразумевает взаимодействие в текстовом виде через TCP-сокет. Клиент отправляет на сервер текстовый запрос, который содержит: * Команду запроса * Заголовки запроса * Пустую строку - признак окончания заголовков запроса * Передаваемые данные, если они подразумеваются

В ответ сервер должен отправить: * Статус обработки запроса * Заголовки ответа * Пустую строку - признак окончания заголовков ответа * Передаваемые данные, если они подразумеваются

Стандартным портом для `HTTP` является порт 80, для `HTTPS` - порт с номером 443, но это жёстко не регламентировано, и при необходимости номер порта может быть любым.

Основные команды и заголовки HTTP

- `GET` - получить содержимое по указанному URL;
- `HEAD` - получить только метаинформацию (заголовки) по указанному URL, но не содержимое;
- `POST` - отправить данные на сервер и получить ответ.

Кроме основных команд, в протоколе HTTP можно определять произвольные дополнительные команды в текстовом виде (естественно, для этого потребуется поддержка как со стороны сервера, так и клиента). Например, расширение WebDAV протокола HTTP, предназначенное для передачи файлов, дополнительно определяет команды `PUT`, `DELETE`, `MKCOL`, `COPY`, `MOVE`.

Заголовки - это строки вида `ключ: значение`, определяющие дополнительную метаинформацию запроса или ответа.

По стандарту `HTTP/1.1`, в любом запросе должен быть как минимум один заголовок - `Host`, определяющий имя сервера. Это связано с тем, что с одним IP-адресом, на котором работает HTTP-сервер, может быть связано много доменных имен.

Полный список заголовков можно посмотреть в Википедии.

Пример взаимодействия:

```
$ telnet ejudge.atp-fivt.org
$ telnet ejudge.atp-fivt.org 80
Trying 87.251.82.74...
Connected to ejudge.atp-fivt.org.
Escape character is '^]'.
GET / HTTP/1.1
Host: ejudge.atp-fivt.org

HTTP/1.1 200 OK
Server: nginx/1.14.0 (Ubuntu)
Date: Tue, 23 Apr 2019 21:18:43 GMT
Content-Type: text/html; charset=UTF-8
Content-Length: 4383
Connection: keep-alive
Last-Modified: Mon, 04 Feb 2019 17:01:28 GMT
ETag: "111f-58114719b3ca3"
Accept-Ranges: bytes

<html>
  <head>
    <meta charset="utf-8"/>
    <title>АКОС ФИВТ МФТИ</title>
  </head>
...
```

Протокол HTTPS

Протокол HTTP**S** - это реализация протокола HTTP поверх дополнительного уровня SSL, который, в свою очередь работает через TCP-сокет. На уровне SSL осуществляется проверка сертификата сервера и обмен ключами шифрования. После этого - начинается обычное взаимодействие по протоколу HTTP в текстовом виде, но это заимодейтвие передается по сети в зашифрованном виде.

Аналогом `telnet` для работы поверх SSL является инструмент `s_client` из состава OpenSSL:

```
$ openssl s_client -connect yandex.ru:443
```

Утилита cURL

Универсальным инструментом для взаимодействия по HTTP в Linux считается curl, которая входит в базовый состав всех дистрибутивов. Работает не только по протоколу HTTP, но и HTTPS.

Основные опции `curl`: * `-v` - отобразить взаимодействие по протоколу HTTP; * `-X КОМАНДА` - отправить вместо `GET` произвольную текстовую команду в запросе; * `-H "Ключ: значение"` - отправить дополнительный заголовок в запросе; таких опций может быть несколько; * `--data-binary "какой-то текст"` - отправить строку в качестве данных (например, для `POST`); * `--data-binary @имя_файла` - отправить в качестве данных содержимое указанного файла.

Библиотека `libcurl`

У утилиты `curl` есть программный API, который можно использовать в качестве библиотеки, не запуская отдельный процесс.

API состоит из двух частей: полнофункциональный асинхронный интерфейс (`multi`), и упрощённый с блокирующим вводом-выводом (`easy`).

Пример использования упрощённого интерфейса:

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

CURL *curl = curl_easy_init();
if(curl) {
  CURLcode res;
  curl_easy_setopt(curl, CURLOPT_URL, "http://example.com");
  res = curl_easy_perform(curl);
  curl_easy_cleanup(curl);
}
```

Этот код эквивалентен команде

```
$ curl http://example.com
```

Дополнительные параметры, эквивалентные отдельным опциям команды `curl`, определяются функцией `curl_easy_setopt`.

Выполнение HTTP-запроса приводит к записи результата на стандартный поток вывода, но обычно бывает нужно получить данные для дальнейшей обработки.

Это делается установкой одной из callback-функций, которая ответственна за вывод:

```c
#include <curl/curl.h>

typedef struct {
  char   *data;
  size_t length;
} buffer_t;

static size_t
callback_function(
            char *ptr, // буфер с прочитанными данными
            size_t chunk_size, // размер фрагмента данных
            size_t nmemb, // количество фрагментов данных
            void *user_data // произвольные данные пользователя
            )
{
  buffer_t *buffer = user_data;
  size_t total_size = chunk_size * nmemb;

  // в предположении, что достаточно места
  memcpy(buffer->data, ptr, total_size);
  buffer->length += total_size;
  return total_size;
}

int main(int argc, char *argv[]) {
  CURL *curl = curl_easy_init();
  if(curl) {
    CURLcode res;

    // регистрация callback-функции записи
    curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, callback_function);

    // указатель &buffer будет передан в callback-функцию
    // параметром void *user_data
    buffer_t buffer;
    buffer.data = calloc(100*1024*1024, 1);
    buffer.length = 0;
    curl_easy_setopt(curl, CURLOPT_WRITEDATA, &buffer);

    curl_easy_setopt(curl, CURLOPT_URL, "http://example.com");
    res = curl_easy_perform(curl);

    // дальше можно что-то делать с данными,
    // прочитанными в buffer

    free(buffer.data);
    curl_easy_cleanup(curl);
  }
}
```

Шифрование с использованием OpenSSL/LibreSSL

Основы шифрования в Linux

Криптография в Linux, как и во многих других UNIX-подобных системах реализована с помощью пакета `openssl` или совместимого с ним форка `libressl`.

Пакет предоставляет: * команду `openssl` для выполнения операций в командной строке * библиотеку `libcrypto` с реализацией алгоритмов шифрования * библиотеку `libssl` с реализацией взаимодействия по протоколам SSL и TLS.

Вычисление хеш-значений

Команды: * `openssl md5` * `openssl sha256` * `openssl sha512`

вычисляют хеш-значение для указанного файла и выводят его в читаемом виде на стандартный поток вывода. Дополнительная опция `-binary` указывает вывод в бинарном формате. Если имя файла не указано, то вычисляется хеш-значение для данных со стандартного потока ввода.

Симметричное шифрование

Команда:

```
openssl enc -ШИФР -in ИМЯФАЙЛА -out ВЫХФАЙЛ
```

Выполяет шифрование *симметричным ключем*, то есть некоторым "паролем", который является одинаковым как для шифрования, так и для обратной операции дешифрования.

Полный список поддерживаемых шифров отображается командой `openssl enc -ciphers`. Наиболее часто используемые:

- `des` - достаточно старый алгоритм с использованием 56-битного ключа;
- `aes256` или `aes-256-cbc` - более надежный и достаточно быстрый;
- `base64` - без шифрования (ключ не требуется); удобный способ конвертировать бинарные файлы в текстовое представление и обратно.

Опция `-d` означает обратное преобразование, то есть *дешифрование*. Опция `-base64` подразумевает, что зашифрованные данные дополнительно преобразуются в кодировку Base64, например, для передачи данных в виде текста.

После запуска команды будет запрошен пароль и его подтверждение. В случае, когда нужно автоматизировать запуск команды, используется опция `-pass`, после которой передается, каким именно образом задается пароль: * `pass:ПАРОЛЬ` - пароль задается обычным текстом в виде аргумента командной строки; жутко небезопасно; * `env:ПЕРЕМЕННАЯ` - пароль задается определенной переменной окружения; немного лучше, но можно выяснить через `/proc/.../environ`; * `file:ИМЯФАЙЛА` - пароль берется из файла; * `fd:ЧИСЛО` - пароль берется из файлового дескриптора с указанным номером; используется при запуске через `fork` + `exec`.

Поскольку алгоритмы симметричного шифрования подразумевают использование ключа фиксированного размера, текстовый пароль произвольной длины предварительно преобразуется с помощью хеш-функции. По умолчанию используется SHA-256, но это можно задавать с помощью опции `-md АЛГОРИТМ`.

Помимо пароля, в ключ входит ещё одна составляющая - *соль* размером 8 байт, которая хранится в самом зашифрованном файле. Это значение генерируется случайным образом, но для возпроизводимости результата может быть явным образом задана с помощью опции `-S HEX`, где `HEX` - восьмибайтное значение в шестнадцатеричной записи.

Шифрование с использованием пары ключей

Стандартным алгоритмом для шифрования с использованием пары ключей считается RSA.

Генерация ключей осуществляется командой:

```
openssl genrsa -out ФАЙЛ РАЗРЯДНОСТЬ
```

Если имя выходного файла не указано, то ключ в текстовом формате будет сохранен на стандартный поток вывода. Обычно ключи RSA хранят в файлах с суффиксом имени `.pem`.

Разрядность определяет стойкость ключа, по умолчанию - 2048 бит.

Поскольку приватный ключ где-то должен храниться, причем безопасным методом, хорошей практикой считается его хранение в зашифрованном виде, для этого используется шифрование с симметричным ключем:

```
openssl genrsa -aes256 -passout ОПЦИИ_ПАРОЛЯ
```

При использовании зашифрованного закрытого ключа, необходимо будет каждый раз указывать пароль, заданный при его создании.

Извлечение публичного ключа из приватного осуществляется командой:

```
openssl rsa -in ПРИВАТНЫЙ_КЛЮЧ -out ПУБЛИЧНЫЙ_КЛЮЧ -pubout
```

Если при создании пары ключей использовалось шифрование, то необходимо ввести пароль, либо задать его через `-passin`.

Шифрование с использованием открытого ключа:

```
openssl rsautl -encrypt -pubin -inkey ПУБЛИЧНЫЙ_КЛЮЧ -in ФАЙЛ -out ВЫХОД
```

Обратная операция с использованием закрытого ключа:

```
openssl rsautl -decrypt -inkey ПРИВАТНЫЙ_КЛЮЧ -in ФАЙЛ -out ВЫХОД
```

Ограничением алгоритма RSA является то, что размер шифруемых данных не может превышать размер ключа. С этим можно бороться следующими способами: 1. Делить исходные данные на блоки размером по 2 или 4 Кбайт и шифровать их по-отдельности 2. Генерировать случаным образом одноразовый *сеансовый ключ*, который будет использован в паре с алгоритмом симметричного шифрования, но сам будет зашифрован с помощью RSA.

```
# Генерируем случайный ключ длиной 30 байт и сохраняем
# его текстовое Base64 представление в переменной $KEY
KEY=`openssl rand -base64 30`


# Шифруем симметричный ключ с помощью открытого ключа RSA
echo $KEY | openssl rsautl -encrypt -pubin \
                           -inkey public.pem \
                           -out symm_key_encrypted.bin

# Шифруем данные из большого файла README.md симметричным
# ключем из переменной $KEY, которая предварительно
# экспортируется, чтобы быть доступной дочернему процессу
export KEY
openssl enc -aes256 -in README.md \
                    -out README_encrypted.bin \
                    -pass env:KEY

# Забываем сеансовый ключ - он больше не нужен
unset KEY
```

Далее можно смело передавать по незащищенному каналу файлы `README_encrypted.bin`, который содержит данные, и `symm_key_encrypted.bin` с зашифрованным симметричным ключем.

Для расшифровки необходимо восстановить сеансовый симметричный ключ, и используя его - дешифровать данные:

```
# Расшифровываем сеансовый симметричный ключ с помощью
# приватного ключа RSA и сохраняем в переменной $KEY
KEY=`openssl rsautl -decrypt \
                    -inkey private.pem \
                    -in symm_key_encrypted.bin`

# Выполняем декодирование файла с данными, используя
# полученный сеансовый ключ
  export KEY
  openssl enc -d -aes256 -pass env:KEY \
              -in README_encrypted.bin

# Забываем расшифрованный сеансовый ключ
unset KEY
```

API библиотеки `libcrypto`

В качестве онлайн-документации по API OpenSSL удобнее использовать документацию из проекта LibreSSL.

Использование с CMake

Если сторонний фреймворк состоит из нескольких библиотек, то команда `find_package` позволяет указать, какие именно необходимо использовать, указав их перечень после `COMPONENTS`:

```
find_package(OpenSSL COMPONENTS Crypto REQUIRED)
```

В случае успешного нахождения `libcrypto` из OpenSSL, будут определены переменные `${OPENSSL_INCLUDE_DIR}` и `${OPENSSL_CRYPTO_LIBRARY}`.

Workflow

Преобразования данных криптографическими функциями подразумевает три стадии: 1. Инициализация - функции, заканчивающиеся на `Init` 2. Добавление очередной порции данных с помощью одной из функций, имя которой заканчивается на `Update` . Этот процесс можно повторять итеративно по мере поступления данных 3. Финализация - функции, оканчивающиеся на `Final` ; на этом этапе появляется итоговый результат преобразования.

Функции `libcrypto`

Функции, имена которых начинаются с `SHA` или `MD5` предназначены для вычисления хеш-значений. Они используют простой workflow из трех стадий.

Для кодирования или декодирования с использованием симметричного ключа используется стандартный workflow, но на стадии инициализации настраивается *контекст* - переменная, которая хранит состояние шифрующего автомата.

Пример:

```
 // Создание контекста
EVP_CIPHER_CTX* ctx = EVP_CIPHER_CTX_new();

// Генерация ключа и начального вектора из
// пароля произвольной длины и 8-байтной соли
EVP_BytesToKey(
  EVP_aes_256_cbc(),    // алгоритм шифрования
  EVP_sha256(),         // алгоритм хеширования пароля
  salt,                 // соль
  password, strlen(password), // пароль
  1,                    // количество итераций хеширования
  key,          // результат: ключ нужной длины
  iv            // результат: начальный вектор нужной длины
);

// Начальная стадия: инициализация
EVP_DecryptInit(
  ctx,                  // контекст для хранения состояния
  EVP_aes_256_cbc(),    // алгоритм шифрования
  key,                  // ключ нужного размера
  iv                    // начальное значение нужного размера
);
```

POSIX API для работы с файловой системой и временем

Каталоги

Каталоги в UNIX-системах - это специальный вид файлов, который содержит набор пар `{inode, name}` для построения иерархии файловой системы.

Как и обычные файлы, каталоги могут быть открыты на чтение или запись с помощью системного вызова `open`. В системе Linux существует не обязательный флаг открытия `O_DIRECTORY`, единственное назначие которого - проверить, что открываемый файл действительно является каталогом, а не файлом другого типа; в противном случае - диагностировать ошибку.

Функции для работы с каталогами

Формат специально файла-каталога зависит от конкретной операционной системы, и абстракцией на уровне POSIX является функции (не системные вызовы!) стандартной библиотеки Си:

```
#include <dirent.h>

// открытие каталога
DIR *opendir(const char *dirname);
DIR *fdopendir(int fd);

// закрытие каталога
int closedir(DIR *dirp);
```

Открытый каталог описывается структурой `DIR`, которая при открытии каталога размещается в куче, и необходимо её освобождать с помощью функции `closedir`.

Чтение из потока каталога осуществляется функцией `readdir`, единицей чтения которой является не байт, а элемент структуры `dirent`:

```
struct dirent {
    ino_t  d_ino;  // inode файла в файловой системе
    char   d_name[NAME_MAX+1]; // имя файла
    /* дальше могут быть ещё какие-нибудь нестандартные поля */
};
```

В системе Linux максимальное имя файла равно 255 (`NAME_MAX`)символам, но при этом, максимальная длина пути к файлу - 4Кб (`PATH_MAX`).

Перемещение текущего указателя чтения записей в каталоге осуществляется с помощью функций `seekdir` и `telldir`.

Каждый каталог, даже пустой, содержит, как минимум, две записи: * специальный файл `.` - каталог, inode которого совпадает с inode того каталога, в котором он содержится; * специальный файл `..` - каталог, inode которого совпадает с inode каталога на уровень выше, либо корневого каталога, если каталог уровнем выше не существует.

Другие функции работы с каталогами:

- `getcwd` - получить текущий рабочий каталог;
- `chdir` - сменить текущий каталог.

'at'-функции

Для того, чтобы упростить работу с относительными путями файлов, в современных версиях `glibc` (Linux и FreeBSD) присуствуют функции для открытия файловых дескрипторов относительно открытого каталога:

```
// Аналоги open
int openat(int dirfd, const char *pathname, int flags);
int openat(int dirfd, const char *pathname, int flags, mode_t mode);

// Аналог stat
int fstatat(int dirfd, const char *pathname, struct stat *statbuf, int flags);
```

Пользователи и группы

Информация о пользователях и группах может храниться как в локальном источнике, например в файлах `/etc/passwd` и `/etc/groups`, так и на удаленных серверах, например LDAP.

Информацию о пользователе или группе можно получить с помощью одной из функций: * `struct passwd *getpwnam(const char *name)` - получить информацию о пользователе по имени; * `struct passwd *getpwuid(uid_t uid)` - получить информацию о пользователе по его User ID; * `struct group *getgrnam(const char *name)` - получить информацию о группе по имени; * `struct group *getgrgid(gid_t gid)` - получить информацию о группе по её Group ID.

Работа со временем

Текущее время

Время в UNIX-системах определяется как количество секунд, прошедшее с 1 января 1970 года, причем часы идут по стандартному гринвичскому времени (GMT) без учета перехода на летнее время (DST - daylight saving time).

32-разрядные системы должны прекратить своё нормальное существование 19 января 2038 года, поскольку будет переполнение знакового целого типа для хранения количества секунд.

Функция `time` возвращает количество секунд с начала эпохи. Аргументом функции (в который можно передать `NULL`) является указатель на переменную, куда требуется записать результат.

В случае, когда требуется более высокая точность, чем 1 секунда, можно использовать системный вызов `gettimeofday`, который позволяет получить текущее время в виде структуры:

```
struct timeval {
  time_t      tv_sec;  // секунды
  suseconds_t tv_usec; // микросекунды
};
```

В этом случае, несмотря на то, что в структуре определено поле для микросекунд, реальная точность будет составлять порядка 10-20 миллисекунд для Linux.

Более высокую точность можно получить с помощью системного вызова `clock_gettime`.

Разложение времени на составляющие

Человеко-представимое время состоит из даты (год, месяц, день) и времени суток (часы, минуты, секунды).

Это описывается структурой:

```
struct tm { /* время, разбитое на составляющие */
  int tm_sec; /* секунды от начала минуты: [0 -60] */
  int tm_min; /* минуты от начала часа: [0 - 59] */
  int tm_hour; /* часы от полуночи: [0 - 23] */
  int tm_mday; /* дни от начала месяца: [1 - 31] */
  int tm_mon; /* месяцы с января: [0 - 11] */
  int tm_year; /* годы с 1900 года */
  int tm_wday; /* дни с воскресенья: [0 - 6] */
  int tm_yday; /* дни от начала года (1 января): [0 - 365] */
  int tm_isdst; /* флаг перехода на летнее время: <0, 0, >0 */
};
```

Для преобразования человеко-читаемого времени в машинное используется функция `mktime`, а в обратную сторону - одной из функций: `gmtime` или `localtime`.

Daylight Saving Time

Во многих странах используется "летнее время", когда стрелки часов переводятся на час назад.

История введения/отмены летнего времени, и его периоды хранится в базе данных IANA.

База данных представляет собой набор правил в тектовом виде, которые компилируются в бинарное представление, используемое библиотекой glibc. Наборы файлов с правилами перехода на летнее время для разных регионов хранятся в `/usr/share/zoneinfo/`.

Когда значение `tm_isdst` положительное, то применяется летнее время, значение `tm_isdst` - зимнее. В случае, когда значение `tm_isdst` отрицательно, - используются данные из timezone data.

Reentrant-функции

Многие функции POSIX API разрабатывались во времена однопроцессорных систем. Это может приводить к разным неприятным последствиям:

```
struct tm * tm_1 = localtime(NULL);
struct tm * tm_2 = localtime(NULL); // opps! *tm_1 changed!
```

Проблема заключается в том, что некоторые функции, например `localtime`, возвращает указатель на структуру-результат, а не скалярное значение. При этом, сами данные структуры не требуется удалять, - они хранятся в `.data`-области библиотеки glibc.

Проблема решается введением *повторно входимых (reentrant)* функций, которые в обязательном порядке трубуют в качестве одного из аргументов указатель на место в памяти для размещения результата:

```
struct tm tm_1; localtime_r(NULL, &tm_1);
struct tm tm_2; localtime_r(NULL, &tm_2); // OK
```

Использование повторно входимых функций является обязательным (но не достаточным) условием при написании многопоточных программ.

Некоторые reentrant-функции уже не актуальны в современных версиях glibc для Linux, и помечены как deprecated. Например, реализация `readdir` использует локальное для каждого потока хранение данных.

Реализация файловых систем без написания модулей ядра

Общие сведения

Файловые системы обычно реализуются в виде модулей ядра, которые работают в адресном пространстве ядра.

Монтирование осуществляется командой `mount(8)`, которой необходимо указать:

- *точку монтирования* - каталог в виртуальной файловой системе, в котором будет доступно содержимое смонтированной файловой системы;
- *тип файловой системы* - один из поддерживаемых типов: `ext2`, `vfat` и др. Если не указать тип файловой системы, то ядро попытается автоматически определить её тип, но сделать это ему не всегда удаётся;
- *устройство для монтирования* - как правило, блочное, устройство для монтирования реальных устройств, либо URI для сетевых ресурсов, либо имя файла для монтирования образа.

Вызов команды `mount` без параметров отображает список примонтированных файловых систем.

Постоянные файловые системы, которые монтируются при загрузке системы, перечислены в файле `/etc/fstab`, формат которого описан в `fstab(5)`. Если точка монтирования указана в этом файле, то для монтирования файловой системы достаточно указать команде `mount` только точку монтирования.

Некоторые типы файловых систем реализованы в виде сервисов, которые работают в пространстве пользователя, как обычные процессы. Ядро взаимодействует с ними, используя файл символьного устройства `/dev/fuse`. Когда ядру необходимо обслужить запрос к виртуальной файловой системе, то в случае, если точка монтирования содержит файловую систему FUSE, ядро формирует запрос в специальном формате, и отправляет его тому процессу, который открыл файл `/dev/fuse` и зарегистрировал открытый файловый дескриптор в качестве параметра системного вызова `mount(2)`. После этого процесс обязан сформировать ответ, который будет обработан модулем ядра `fuse.ko` и ядро выполнит запрошенную файловую операцию.

Подсистема FUSE реализована в Linux и FreeBSD.

Пример реализации - файловая система SSH

Протокол SSH предназначен для терминального доступа к любой UNIX-системе, на которой запущен сервис `sshd`. Авторизация осуществляется через пароль, в этом случае нужно будет его ввести после запуска команды `ssh`, либо с использованием асимметричных RSA-ключей.

Использование ключей обычно является более безопасным (при условии, что полностью запрещена авторизация по паролю), поскольку случайно подобранный ключ намного сложнее подобрать по словарю, чем пароль. Для создания пары ключей используется команда `ssh-keygen`, которая создает пару файлов `~/.ssh/id_rsa` и `~/.ssh/id_rsa.pub`, первый из которых является приватным ключом, а второй - публичным. Содержимое публичного ключа можно добавить отдельной строкой в текстовый файл `~/.ssh/authorized_keys` на целевом хосте, и после этого можно будет подключаться без ввода пароля. Для копирования ключа на удаленный сервер в большинстве дистрибутивов предусмотрен скрипт `ssh-copy-id`.

С помощью ssh можно не только интерактивно взаимодействовать с удаленным хостом, но и выполнять отдельные команды, если указывать из последними аргументами.

Пример: создание каталога на удаленном хосте и копирование в него файла с локального компьютера.

```
# ssh user  @ host    "command to execute"
> ssh victor@10.0.2.4 "mkdir ~/some_dir"

# get contents   pipe                    write contents to file
> cat /bin/bash   |  ssh victor@10.0.2.4 "cat >~/some_dir/bash"

# set file attributes
> ssh victor@10.0.2.4 "chmod a+x ~/some_dir/bash"

# ensure file has been copied
> ssh ssh victor@10.0.2.4 "ls -l ~/some_dir/"
total 1024
-rwxr-xr-x  1 victor  victor  1012552 Apr 21 10:17 bash
```

Таким образом, используя ssh в сочетании со стандартными командами POSIX, можно реализовать произвольные файловые операции над удаленной файловой системой. Этот подход реализован в реализации файловой системы sshfs.

```
# local directory for remote contents
> mkdir remote_host

# mount   user@host    :path    local mount point
> sshfs victor@10.0.2.4:/       remote_host
```

При этом, реализация `sshfs` использует только `fork+exec` для запуска команды `ssh`, и работает в адресном пространстве пользователя, а не ядра, и использует обычные сторонние библиотеки:

```
> ldd /usr/bin/sshfs
        linux-vdso.so.1 (0x00007ffff1985000)
        libfuse.so.2 => /lib64/libfuse.so.2 (0x00007f33e781e000)
        libgthread-2.0.so.0 => /usr/lib64/libgthread-2.0.so.0 (0x00007f33e761c000)
        libglib-2.0.so.0 => /usr/lib64/libglib-2.0.so.0 (0x00007f33e7305000)
        libpthread.so.0 => /lib64/libpthread.so.0 (0x00007f33e70e7000)
        libc.so.6 => /lib64/libc.so.6 (0x00007f33e6d2d000)
        libdl.so.2 => /lib64/libdl.so.2 (0x00007f33e6b29000)
        libpcre.so.1 => /usr/lib64/libpcre.so.1 (0x00007f33e689c000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f33e7c71000)
```

Библиотека libfuse

Реализация демона FUSE

Библиотека libfuse реализует функциональность взаимодействия с модулем ядра `fuse.ko` через файловый дескриптор файла `/dev/fuse`. Поскольку проект существует достаточно давно, программный интерфейс (API) библиотеки претерпел множество изменений, и перед включением заголовочного файла необходимо указать версию программного интерфейса, который будет использован:

```
#define FUSE_USE_VERSION 30   // API version 3.0
#include <fuse.h>
```

В дальнейшем значение макроса `FUSE_USE_VERSION` будет использовано препроцессором при обработке файла `fuse.h` для условной подстановки соответствующих определенной версии сигнатур функций. До версии 3.0 изменений накопилось очень много, поэтому используется отдельная библиотека `libfuse3.so` вместо `libfuse.so`, в дальнейшем мы будем использовать именно её.

Для FUSE не реализован пакет CMake, но существует описание в формате `pkg-config`, которое можно использовать из CMake:

```
find_package(PkgConfig REQUIRED)
pkg_check_modules(FUSE REQUIRED fuse3)

link_libraries(${FUSE_LIBRARIES})         # -lfuse3 -lpthread
include_directories(${FUSE_INCLUDE_DIRS}) # -I/usr/include/fuse3
compile_options(${FUSE_CFLAGS_OTHER})     # empty since fuse 3.0
```

Реализация файловой системы - это программа-демон, которая ожидает запросы от ядра и обслуживает их. Реализация тривиальной программы:

```
static struct fuse_operations operations = {
    // pointers to callback functions
};

int main(int argc, char *argv[]) {
    // arguments to be preprocessed before passing to /sbin/mount.fuse3
    struct fuse_args args = FUSE_ARGS_INIT(argc, argv);

    // run daemon
    int ret = fuse_main(
        args.argc, args.argv, // arguments to be passed to /sbin/mount.fuse3
        &operations,          // pointer to callback functions
        NULL                  // optional pointer to user-defined data
    );
    return ret;
}
```

Демон использует стандартный для монтирования набор аргументов командной строки, аналогичный команде `mount.fuse(8)`. Как минимум, требуется один позиционный аргумент - это точка монтирования. После выполнения операции монтирования, демон продолжает работать в фоновом режиме, используя `fork`. Для работы в текущем процессе (foreground) используется опция `-f`, что бывает полезно для отладки. Опция `-u` означает операцию отключения ранее зарегистрированной точки монтирования.

Набор операций, который используется при обработке запросов от ядра может быть не полным, то есть не покрывать всю функциональность файловой системы, или даже пустым. В этом случае, при попытке обращения к файловой системе, возникнет ошибка. Так, для примера выше, команда `ls` завершит работу с ошибкой:

```
# start empty FUSE implementation
> ./my_filesystem work_dir

# try to get filesystem contents
> ls work_dir
ls: cannot access 'work_dir': Function not implemented

# umount filesystem and stop the daemon
> fusermount3 -u work_dir
```

Реализация функциональности

Реализация функциональности файловой системы определяется указателями на соответствующий функции в структуре `struct fuse_operations`, причем для большинства полей имена совпадают с именами соответствующих системных вызовов (за исключением системного вызова `stat`, поле которого называется `getattr`). Для большинства из функций возвращаемым значением является целое число: `0` в случае успеха, и отрицательное значение в случае ошибки. Значение модуля кода ошибки соответствует ожидаемому коду ошибки, который будет записан в `errno` после выполнения соответствующего системного вызова. Так, ошибке "файл не найден" соответствует возвращаемое значение `-ENOENT`, где значение константы `ENOENT` определено в заголовочном файле `<errno.h>`.

Полное описание callback-функций доступно по ссылке: https://libfuse.github.io/doxygen/structfuse__operations.html.

Получение списка файлов

Рассмотрим тривиальную файловую систему, которая содержит ровно два файла: `a.txt` и `b.txt` с одинаковым содержимым. Как минимум, необходимо иметь возможность узнать содержимое файловой системы, реализовав для этого `readdir(2)` для чтения содержимого каталога, и `stat(2)` для получения атрибутов файлов, включая атрибуты самого корневого каталога, иначе будет невозможно узнать о том, что он действительно является каталогом, и тем самым - получить его содержимое.

```
 // contents to be accessed by reading files
static const char DummyData[] = "Hello, World!\n";

// callback function to be called after 'stat' system call
int my_stat(const char *path, struct stat *st, struct fuse_file_info *fi)
{
    // check if accessing root directory
    if (0==strcmp("/", path)) {
        st->st_mode = 0555 | S_IFDIR;  // file type - dir, access read only
        st->st_nlink = 2;              // at least 2 links: '.' and parent
        return 0;                      // success!
    }
    if (0!=strcmp("/a.txt", path) && 0!=strcmp("/b.txt", path)) {
        return -ENOENT; // error: we have no files other than a.txt and b.txt
    }

    st->st_mode = S_IFREG | 0444;    // file type - regular, access read only
    st->st_nlink = 1;                // one link to file
    st->st_size = sizeof(DummyData); // bytes available
    return 0;                        // success!
}

// callback function to be called after 'readdir' system call
int my_readdir(const char *path, void *out, fuse_fill_dir_t filler, off_t off,
               struct fuse_file_info *fi, enum fuse_readdir_flags flags)
{
    if (0 != strcmp(path, "/")) {
        return -ENOENT;  // we do not have subdirectories
    }

    // two mandatory entries: the directory itself and its parent
    filler(out, ".", NULL, 0, 0);
    filler(out, "..", NULL, 0, 0);

    // directory contents
    filler(out, "a.txt", NULL, 0, 0);
    filler(out, "b.txt", NULL, 0, 0);

    return 0;  // success
}

struct fuse_operations operations = {
    .readdir = my_readdir,   // callback function pointer for 'readdir'
    .getattr = my_stat,      // callback function pointer for 'stat'
};
```

Теперь реализация файловой системы позволяет получить содержимое каталога:

```
 > ./my_filesystem work_dir

> ls -l work_dir
total 0
-r--r--r-- 1 root root 15 Jan  1  1970 a.txt
-r--r--r-- 1 root root 15 Jan  1  1970 b.txt

# try to get file contents - still not implemented 'open' and 'read'
> cat work_dir/a.txt
cat: work_dir/a.txt: Function not implemented

# umount filesystem and stop the daemon
> fusermount3 -u work_dir
```

Обратите внимание, что даты создания файлов - 1 января 1970 года, - это соответствует значению 0 для формата времени в UNIX, а владелец файла и группа - пользователь и группа `root`, численные значения `uid` и `gid` которых равны 0. Эти поля могут быть также заполнены при реализации `stat`.

Кроме того, утилита `ls` отображает `total 0`, поскольку это значение в выводе является количеством занятых блоков на диске, и эта информация отсутствует в атрибутах файлов.

Чтение данных

Для того, чтобы прочитать данные из файла, его нужно, как минимум, успешно открыть, и кроме того, реализовать функцию чтения, которая соответствует поведению системного вызова `read`.

```
// callback function to be called after 'open' system call
int my_open(const char *path, struct fuse_file_info *fi)
{
    if (0!=strcmp("/a.txt", path) && 0!=strcmp("/b.txt", path)) {
        return -ENOENT;   // we have only two files in out filesystem
    }
    if (O_RDONLY != (fi->flags & O_ACCMODE)) {
        return -EACCES;   // file system is read-only, so can't write
    }
    return 0;  // success!
}

// contents of file
static const char DummyData[] = "Hello, World!\n";

// callback function to be called after 'read' system call
int my_read(const char *path, char *out, size_t size, off_t off,
         struct fuse_file_info *fi)
{
    // 'read' might be called with arbitary arguments, so check them
    if (off > sizeof(DummyData))
        return 0;
    // reading might be called within some non-zero offset
    if (off+size > sizeof(DummyData))
        size = sizeof(DummyData) - off;
    const void *data = DummyData + off;
    // copy contents into the buffer to be filled by 'read' system call
    memcpy(out, data, size);
    // return value is bytes count (0 or positive) or an error (negative)
    return size;
}

// register functions as callbacks
struct fuse_operations operations = {
    .readdir = my_readdir,
    .getattr = my_stat,
    .open    = my_open,
    .read    = my_read,
};
```

Теперь можно прочитать содержимое файла:

```
 > ./my_filesystem work_dir

# get file contents - OK
> cat work_dir/a.txt
Hello, World!

# try to create new file - still not implemented
> touch work_dir/new_file.txt
touch: cannot touch 'work_dir/new_file.txt': Function not implemented

# umount filesystem and stop the daemon
> fusermount3 -u work_dir
```

Опции монтирования

У монтирования файловых систем могут быть опции, например точка монтирования и файл с образом, либо устройство, либо любой другой источник данных для файловой системы. Некоторые опции являются обязательными для всех FUSE-систем, например указание точки монтирования, а часть из них - быть специфичными для реализации определенных файловых систем.

Функция, реализующая работу FUSE-демона `fuse_main` принимает два аргумента: количество опций `argc` и массив строк `argv`, по аналогии с функцией `main`. Если какие-то опции не распознаны `fuse_main`, либо их не достаточно для монтирования файловой системы, то эта функция завершается с ошибкой.

Для выделения, специфичных для конкретной файловой системы, опций используется модифицируемый список опций `fuse_args`, который инициализируется макросом `FUSE_ARGS_INIT(argc, argv)`.

Для извлечения специфичных опций по некоторым шаблонам используется функция `fuse_opt_parse`, которая принимает описания опций, которые необходимо распознать, выполняет разбор аргументов командной строки, и извлекает обработанные опции из массива `argv` структуры `fuse_args`, чтобы они потом не попали в `fuse_main`.

Описанием одной опции является структура `fuse_opt`, которая содержит:

- текстовую строку, содержащую шаблон аргумента, и возможно, форматную строку для значения, которое необходимо извлечь;
- смещение в байтах относительно начала структуры, которую заполняет функция `fuse_opt_parse`, если параметр встретился среди аргументов командной строки;
- целочисленное значение, которое будет записано в структуру; игнорируется в случае, если шаблон содержит формат значения, который нужно извлечь.

Для разбора опций необходимо определить массив из структур `fuse_opt`, где последний элемент, по аналогии со строками, заполнен нулями, и вызвать `fuse_opt_parse`, передав указатель на структуру с опциями, которую необходимо заполнить по результатам их разбора.

```
int main(int argc, char *argv[])
{
    // initialize modificable array {argc, argv}
    struct fuse_args args = FUSE_ARGS_INIT(argc, argv);

    // struct to be filled by options parsing
    typedef struct {
        char *src;
        int  help;
    } my_options_t;

    my_options_t my_options;
    memset(&my_options, 0, sizeof(my_options));

    // options specifications
    struct fuse_opt opt_specs[] = {
        // pattern: match --src then string
        // the string value to be written to my_options_t.src
        { "--src %s", offsetof(my_options_t, src)   , 0     },
        // pattern: match --help
        // if found, 'true' value to be written to my_options_t.help
        { "--help"  , offsetof(my_options_t, help)  , true  },
        // end-of-array: all zeroes value
        { NULL      , 0                             , 0     }
    };

    // parse command line arguments, store matched by 'opt_specs'
    // options to 'my_options' value and remove them from {argc, argv}
    fuse_opt_parse(&args, &my_options, opt_specs, NULL);

    if (my_options.help) {
        show_help_and_exit();
    }

    if (my_options.src) {
        open_filesystem(my_options.src);
    }

    // pass rest options but excluding --src and --help to mount.fuse3
    int ret = fuse_main(args.argc, args.argv, &operations, NULL);

    return ret;
}
```

Время в UNIX

API для работы со временем

Текущее время

Время в UNIX-системах определяется как количество секунд, прошедшее с 1 января 1970 года, причем часы идут по стандартному гринвичскому времени (GMT) без учета перехода на летнее время (DST - daylight saving time).

32-разрядные системы должны прекратить своё нормальное существование 19 января 2038 года, поскольку будет переполнение знакового целого типа для хранения количества секунд.

Функция `time` возвращает количество секунд с начала эпохи. Аргументом функции (в который можно передать `NULL`) является указатель на переменную, куда требуется записать результат.

В случае, когда требуется более высокая точность, чем 1 секунда, можно использовать системный вызов `gettimeofday`, который позволяет получить текущее время в виде структуры:

```
struct timeval {
  time_t      tv_sec;  // секунды
  suseconds_t tv_usec; // микросекунды
};
```

В этом случае, несмотря на то, что в структуре определено поле для микросекунд, реальная точность будет составлять порядка 10-20 миллисекунд для Linux.

Более высокую точность можно получить с помощью системного вызова `clock_gettime`.

Разложение времени на составляющие

Человеко-представимое время состоит из даты (год, месяц, день) и времени суток (часы, минуты, секунды).

Это описывается структурой:

```
struct tm { /* время, разбитое на составляющие */
  int tm_sec; /* секунды от начала минуты: [0 -60] */
  int tm_min; /* минуты от начала часа: [0 - 59] */
  int tm_hour; /* часы от полуночи: [0 - 23] */
  int tm_mday; /* дни от начала месяца: [1 - 31] */
  int tm_mon; /* месяцы с января: [0 - 11] */
  int tm_year; /* годы с 1900 года */
  int tm_wday; /* дни с воскресенья: [0 - 6] */
  int tm_yday; /* дни от начала года (1 января): [0 - 365] */
  int tm_isdst; /* флаг перехода на летнее время: <0, 0, >0 */
};
```

Для преобразования человеко-читаемого времени в машинное используется функция `mktime`, а в обратную сторону - одной из функций: `gmtime` или `localtime`.

Daylight Saving Time

Во многих странах используется "летнее время", когда стрелки часов переводятся на час назад.

История введения/отмены летнего времени, и его периоды хранится в базе данных IANA.

База данных представляет собой набор правил в текстовом виде, которые компилируются в бинарное представление, используемое библиотекой glibc. Наборы файлов с правилами перехода на летнее время для разных регионов хранятся в `/usr/share/zoneinfo/`.

Когда значение `tm_isdst` положительное, то применяется летнее время, значение `tm_isdst` - зимнее. В случае, когда значение `tm_isdst` отрицательно, - используются данные из timezone data.

Reentrant-функции

Многие функции POSIX API разрабатывались во времена однопроцессорных систем. Это может приводить к разным неприятным последствиям:

```
struct tm * tm_1 = localtime(NULL);
struct tm * tm_2 = localtime(NULL); // opps! *tm_1 changed!
```

Проблема заключается в том, что некоторые функции, например `localtime`, возвращает указатель на структуру-результат, а не скалярное значение. При этом, сами данные структуры не требуется удалять, - они хранятся в `.data`-области библиотеки glibc.

Проблема решается введением *повторно входимых (reentrant)* функций, которые в обязательном порядке требуют в качестве одного из аргументов указатель на место в памяти для размещения результата:

```
struct tm tm_1; localtime_r(NULL, &tm_1);
struct tm tm_2; localtime_r(NULL, &tm_2); // OK
```

Использование повторно входимых функций является обязательным (но не достаточным) условием при написании многопоточных программ.

Некоторые reentrant-функции уже не актуальны в современных версиях glibc для Linux, и помечены как deprecated. Например, реализация `readdir` использует локальное для каждого потока хранение данных.

Виды часов

Время в UNIX системе представляется в виде двух чисел: количества секунд и количества наносекунд, но это не означает, что точность часов сопоставимо с одной наносекундой. В современных компьютерах архитектуры несколько типов часов:

- аппаратные часы, которые работают от отдельной батарейки даже при отключения питания;
- счетчик в ядре операционной системы, который периодически обновляется отдельным аппаратным таймером;
- счетчик тактов процессора.

Аппаратные часы

Аппаратные часы обычно работают на базе стандартного часового кварца, обеспечивающего частоту 32.768КГц, и имеющие точность, сопоставимую с точностью обычных бытовых часов. Эти часы могут хранить дату как в формате UTC (стандарт, принятый в UNIX-системах), так и локальное время (принято в Windows).

В Linux аппаратные часы доступны в виде символьного устройства `/dev/rtc`, доступ к которому есть только у пользователя `root`.

Это устройство может быть открыто только для чтения, после чего из него можно читать 32-битные значения - информацию о прерываниях. Настройка поведения часов осуществляется с помощью системного вызова `ioctl` и передачей одной из команд, относящихся к `rtc(4)`.

Прерывания могут быть:

- каждую секунду, если часы настроены на ежесекундное срабатывание `RTC_UIE`
- с частотой от 2 до 8192 Гц, причем частота должна быть степенью двойки `RTC_PIE`
- срабатывание в определенное время `RTC_AIE`.

Ежесекундное прерывание с использованием часов реального времени:

```
#include <sys/ioctl.h>       // системный вызов ioctl
#include <linux/rtc.h>       // константы RTC_*

int rtc = open("/dev/rtc", O_RDONLY);
if (-1==rtc) { perror("open /dev/rtc"); exit(1); } // только root может открыть

ioctl(rtc, RTC_UIE_ON, 0);   // включаем прерывания каждую секунду
while (1) {
    int interrupt_mask;
    // системный вызов read блокируется до следующего прерывания
    read(rtc, &interrupt_mask, sizeof(interrupt_mask));
    puts("Tick");
}
```

Аппаратные часы хранят информацию о текущей дате и текущем времени с точностью до секунды, причем это время может отличаться от системного.

Получение времени из аппаратных часов:

```
#include <sys/ioctl.h>       // системный вызов ioctl
#include <linux/rtc.h>       // константы RTC_*, а ещё структура rtc_time

int rtc = open("/dev/rtc", O_RDONLY);
if (-1 == rtc) { perror("open /dev/rtc"); exit(1); }

struct rtc_time t = {};
// чтение текущего времени из аппаратных часов
ioctl(rtc, RTC_RD_TIME, &t);

printf("RTC time: %02d : %02d : %02d \n",
       t.tm_hour, t.tm_min, t.tm_sec);
```

Синхронизация системного времени с аппаратными часами осуществляется при загрузке системы и завершении работы (выключении или перезагрузке).

Команда `hwclock` позволяет взаимодействовать с часами реального времени, в том числе и для синхронизации.

```
> hwclock -r     # прочитать и вывести время из RTC
> hwclock -w     # сохранить системное время в RTC (обычно при выключении)
> hwclock -s     # установить системное время из RTC (при загрузке)
```

По умолчанию подразумевается, что аппаратные часы используют время UTC, но можно если компьютер используется совместно с системой Windows (двойная загрузка), то можно синхронизировать локальное время, для этого используется опция `-l`. Многие дистрибутивы Linux на этапе установки позволяют указать, какое именно время хранится в часах реального времени.

Системное время и источники времени

Отсчет времени начинается с момента загрузки ядра, и хранится в виде целого количества *тиков* (jiffies), продолжительность которых определяется параметром компиляции ядра `CONFIG_HZ`, и может принимать одно из значений: 100, 250, 300 или 1000 Гц. Для текущего ядра это можно выяснить в файле `/boot/config-ВЕРСИЯ_ЯДРА`.

Более высокая частота подразумевает большую нагрузку на процессор, но бывает полезна в некоторых применениях, когда требуется повысить отзывчивость системы.

Доступные источники времени зависят от архитектуры процессора и конфигурации ядра, узнать в Linux их можно командой:

```
> cat /sys/devices/system/clocksource/clocksource0/available_clocksource
tsc hpet acpi_pm
#^   ^    ^
#|   |    Legacy-драйвер
#|   Системный таймер высокой точнсти, обычно работает на частоте от 10Мгц
#Регистр Time-Step Counter в самом процессоре
```

Текущий способ определения точного времени хранится в `current_clocksource`:

```
> cat /sys/devices/system/clocksource/clocksource0/current_clocksource
tsc
```

Наиболее точным источником времени, в то же время с минимальным временем доступа, - это счетчик тактов в самом процессоре Time-Step Counter, значение которого, для архитектуры x86 можно получить с помощью команды `RDTSC`. Поскольку получение высокоточных значений времени используется при эксплуатации уязвимостей процессоров Meltdown и Spectre, то этот способ может быть принудительно отключен в системе.

Для получения текущего времени из источника времени используется системный вызов `clock_gettime`:

```
#include <time.h>

struct timespec {
    time_t tv_sec;  // время в секундах
    long   tv_nsec; // доля времени в наносекундах
};

int clock_gettime(clockid_t id, /* out: */ struct timespec *tp);
```

Первый параметр системного вызова - это целочисленное значение, определяющее, какой именно счетчик или таймер нужно использовать. Для большинства UNIX-систем определены таймеры:

- `CLOCK_REAL` - значение астрономического времени, где за точку отсчета принимается *начало эпохи* - 1 января 1970 года;
- `CLOCK_MONOTONIC` - значение времени с момента загрузки ядра, исключая то время, пока система находилась в спящем режиме;
- `CLOCK_PROCESS_CPUTIME_ID` - значение времени, затраченного на выполнение текущего процесса;
- `CLOCK_THREAD_CPUTIME_ID` - значение времени, затраченного на выполнение текущего потока.

Этот системный вызов в FreeBSD, и ядре Linux до версии 2.6.21, для стандартных таймеров возвращает значение, которое было обновлено в момент предыдущего аппаратного прерывания от системного таймера, то есть точность времени не превышает продолжительности одного тика.

В современных версиях Linux происходит обращение к регистру TSC, либо опрос системного таймера, который возвращает текущее значение с высокой точностью. Часы `CLOCK_REAL_COARSE` и `CLOCK_MONOTONIC_COARSE` возвращают время с точностью до одного тика, как в старых версиях.

В системе FreeBSD предусмотрены два вида часов - точные, с суффиксом `_PRECISE`, которые опрашивают системный таймер, и быстрые, с суффиксом `_FAST`, которые возвращают значения с точностью до тика. POSIX-совместимым названиям часов соответствуют `_FAST` -версии.

Системный вызов `clock_gettime` реализован в виде `vdso(7)` -функции, которая доступна в адресном пространстве пользователя. В случае, если происходит опрос часов с низкой точностью (например `CLOCK_REAL_COARSE` в Linux или `CLOCK_REAL_FAST` в FreeBSD), то время вычисляется в адресном пространстве пользователя по значению из счетчика, ранее проставленного планировщиком задач. Если же требуется получить время с высокой точностью и не используется TSC, то может потребоваться настоящий системный вызов для опроса системного таймера.