Unix-системы

10 февраля 2017

UNIX-like:

- Linux
- BSD
- Solaris
- AIX
- HPUX

Пакеты, обычно представляющий из себя некоторый архив) установки в UNIX состоят из нескольких частей:

1. Метаинформация;
2. Скрипты по «монтажу» и «демонтажу» ПО;
3. Данные пакета

Бывают разные пакеты. Бинарные пакеты уже содержат некоторый исполняемый код, который уже готов к запуску и использованию.

Также пакеты могут содержать лишь исходники, которые компилируются уже на целевой машине. Они могут как быть скомпилированы автоматически в процессе «монтажа» и установлены, так и собраны в бинарные пакеты (один или несколько).

1. Пакетный менеджер
    - .deb — бинарные пакеты (обрабатывается пакетным менеджером dpkg)
    - .rpm — бинарные пакеты (обрабатывается пакетным менеджером rpm)
    - .tgz и другие — исходники, «сорцовые» пакеты. В Gentoo для их обработки используется программа emerge, которая отслеживает корректность компиляции и приминения изменений в систему.
    - .srpm — формат «сорцового» пакета, который после компиляции даст набор .rmp пакетов, которые можно обработать при помощи rpm.

Дистрибутивы, основанные на deb:

- Debian
- Ubuntu
- Mint
- ...и многие другие...

Дистрибутивы, основанные на rpm:

- Fedora (к слову, платная)
- CentOS (тоже самое, но бесплатная :))
- Suse
- ...

Дистрибутивы, основанные на исходниках:

- Gentoo
- Slackware
- ...

Раньше установка дистрибутива представляла из себя установку базы и выбора набора пакетов, необходимых именно Вам.

Сейчас же используются *репозитории* (У Debian, например, сетевой репозиторий расположен по адресу ftp.debian.org). Репозиторий представляет из себя набор *веток* пакетов. Ветка характеризуется тем, что пакеты в ней образуют в некотором смысле согласованное множество, т.е. все зависимости этих пакетов не выходят за границы этой ветки.

Для работы с репозиториями существует специальный набор программ. У deb-систем этим занимается программный набор **apt**, а конкретно **apt-get** и **aptitude**. С ними доступны следующие опции:

- **update**. Эта опция позволяет прочитать список пакетов с репозитория, синхронизируя вышу информацию о наборе пакетов с информацией репозитория.
- **upgrade**. Он обновляет пакеты, которые уже стоят на компьютере. **dist-upgrade** к тому же будет сносить всё, что ему тем или иным образом мешает.
- **install**. Эта опция устанавливает нужный пакет, а так же необходимые зависимости, возможно, предлагая какие-то доп.опции.
- **remove**. Удаляет пакет вместе со всеми зависимости, которые становятся невостребованными в системе

Существует также программа **apt-cache**. Если добавить опцию **show**, то будет отображена информация по пакету. **search** позволяет узнать, есть ли в репозитории(ях) пакет, который ищется по заданным ключевым словам.

Пакеты, необходимые для работы в семестре

Редактирование файлов:

- vim
- vim-scripts
- vim-runtime

Форматирование файлов:

- astyle. Форматирование кода через `astyle --style=ansi /file/path/prog.c`
- kdiff3
- meld. Вместе с предыдущим пакетом показывают разницу между файлами.

Файловые менеджеры:

- mc. Он же midnight commander

Разработка и отладка:

- gcc — компилятор C
- gdb — GNU Debugger. Графические оболочки ddd и kgdb.
- valgrind — отладчик памяти
- gprof — следит за временем
- glibc-doc, manpages-dev, gcc-doc — загрузка доп ресурсов для man. Для перегонки man в pdf есть arps.
- make

Работа с репозиториями:

- git

Средства удалённого доступа:

- ssh

Просмотр и редактирование:

- LibreOffice (совместимость с MS Office)
- Что-нибудь для pdf (evince, okular, xpdf)

Браузер

Графический интерфейс:

- xorg — основа для графического интерфейса
- Оконный менеджер:
  - icewm, ctwm
  - xfce4
  - Gnome, KDE.

Информация о репозиториях лежит в файле `/etc/apt/sources.list`

Установка дистрибутива Linux

1. Выбор региональных параметров
2. Разметка диска:
   - / — все системные утилиты
   - /usr — пакеты, которые нужны больше пользователю, чем системе
   - /var
   - /home — домашний пользовательский каталог
   - /tmp
   - /usr/local — размер зависит от того, сколько доп.софта будете ставить

Информация о точках монтирования лежит в файле `/etc/fstab`

Как при установке не вляпаться в винду

У винды есть следующие особенности:

- EFI-раздел. Сюда загрузчик складывает образ загрузчика, обращающийся к boot.
- Скрытый раздел с системными утилитами
- Опционально: раздел восстановления

Вот этих товарищей трогать не надо.

Программы на C

C-программа — набор файлов с расширениями `.c` и `.h` . Сборка состоит из следующих шагов:

1. Макроподстановка, замена всех макросов на то, что он собой символизирует.
2. Компиляция в assembler.
3. Преобразование в объектный код.
4. Линковка — связка отдельных объектных файлов в единую программу.

.h файлы в результате компиляции должны выдавать пустой объектный код.

Компиляторы

В мире UNIX принят компилятор `cc` — то же самое, что и `gcc`. У компиляторов есть различные ключи:

- `-E` — остановиться на стадии прекомпиляции и вывести в стандартный поток получившийся код.
- `-S` — из файла с расширением `.c` сгенерирует `.s` — файл с ассемблерным кодом.
- `-c` — создание файла `.o` с объектным кодом.
- `-I каталог` — поиск подключаемых файлов в каталоге `каталог`. Допустимо множественное использование.
- `-D` — создание макропеременной. `-D ENABLE_CUDA`, например.
- `-L` — список каталогов для поиска библиотек.
- `-l<lib_name>` — подключение библиотеки.
  - Библиотека: файл `lib<name>` с расширением `.a` (статическая библиотека) или `.so` (динамически подключаемая библиотека).
- `-o` — имя выходного файла.
- `-g` — добавляет в объектный файл текст исходного файла
- `-pg` — добавление профилировщика. Перед каждой функцией компилятор добавляет некоторую конструкцию, после считает, сколько раз и сколько времени работала каждая функция.
- `-O` — установка уровня оптимизации. `-O0` — без оптимизации. `-O1` — чуть-чуть оптимизируем. По ассемблерному коду ещё можно восстановить исходник. `-O2` — более агрессивно. `-O3` — очень агрессивная оптимизация. Можно и дальше, но на самом деле — всё фигня.

- `-Os` — оптимизация по размеру. т.е. генерация кода так, чтобы место, занимаемое в оперативной памяти, было минимальным.
- `-march` — задаёт конкретную архитектуру, под которую компилируется код. Например, `-march PPC64` — для Power PC 64.
- `-mtune` — выбор конкретного поколения процессора. Можно указать `native`, в этом случае компилятор сам посмотрит в нужный файл и будет делать код под ваш процессор.
- `--std=<standart_name>` — стандарт компиляции. `-ansi` — компиляция в стандарте C89 (ANSI C).
- `-pedantic` — «педантично» следовать стандарту. Даже компилироваться не будет.
- `-Wall` — вывод всех предупреждений.
- `-Wextra` — ещё больше предупреждений путём продвинутой диагностики.
- `-Werror` — всё, что не нравится компилятору, становится ошибкой.

Секции `man`'а.

1. Программы
2. Системные вызовы
3. Библиотечные функции
4. хз
5. Конфигурационные файлы
6. Xwindow
7. Философская секция
8. Демоны (daemons) и системные утилиты.

C

Базовые типы:

- `char`
- `short`
- `int`
- `long int`
- `long long int`

- `float`
- `double`
- `long double`

Модификаторы типов:

- `unsigned`
- `const` — запихнуть переменную в область кода.
- `static` — описание не на стеке, а в области данных. Можно обращаться только из этой функции. В глобальной области: только из этого файла.
- `extern` — тело переменной находится в другом объектном файле. Найти — задача линковщика.
- `auto` — размещение одтаётся компилятору.
- `register` — просьба компилятору **по возможности** разместить переменную в регистре.
- `volatile` — приказывает переменной всегда находится в одной и той же памяти.

0.1 Ввод-вывод в C

Низкоуровневый:

- open
- read
- write
- ioctl
- ...

Высокоуровневый ввод-вывод реализован через `FILE` (stdio.h).

Используют `FILE*`. С каждой программой ассоциированы `stdin`, `stdout`, `stderr`.

Например: `fopen(<filename>, <format_string>)`, который возвращает `FILE*`, если всё плохо, то NULL.

Строка формата может быть одной из:

- `r` — чтение
- `w` — запись
- `a` — добавление
- `r+` — чтение с возможностью записи

- `w+` — запись с возможностью чтения
- `w+` — добавление с возможностью чтения

Перемещение курсора делается при помощи `fseek`.
Для работы с `FILE*` используют `fread`, `fwrite`.

```
ssize_t fread(void*, size_t num_itmes, size_t item_size, FILE*)
```

Первый аргумент — то, куда мы хотим считать. `fwrite` аналогичен.
`int fgetc(FILE* f)` — читает символ из файла, возвращает как `int`, если что-то пошло не так или файл закончился: `EOF`.
`int ungetc(FILE*, char)` — вернуть символ наверх.
`int putc(int, FILE*)` — положить символ в файл.
`getchar()`, `putchar` — аналоги со стандартными файлами.
Для чтения используются `scanf`, `gets`. Но это опасно. Есть

```
void* fgets(char*, size_t buf_size, FILE*)
```

0.2 Домашка 3

Реализовать

```
int read_string(char **str, FILE*f)
```

или

```
char* read_string(FILE*, int* err_code)
```

Реализация должна быть безопасной.
Работа с динамической памятью:

```
void* malloc(size_t size) //NULL, если всё плохо
void free(void* pointer)
void* calloc(size_t num, size_t size) //num - размер массива, size - размер
   //элемента, инициализация
void* realloc(void* pointer, size_t new_size) //пытается увеличить объем
   //памяти до new_size
```

0.3 Make-файлы

Пусть мы хотим большой программный проект из пачки файлов. Цепочка include'ов — плохой тон, не надо так.
Для этого созданы Make-файлы. На Linux используют maven.
Есть понятия **цель** и **подцели**. Создан некоторый декларативный язык.
Пусть программа состоит из a.c и b.c.

```
all : prog
a.o : a.c
    gcc -c a.c
b.o : b.c
    gcc -c b.c
prog : a.o b.o
    gcc -o prog a.o b.o
clean :
    rm -f prog a.o b.o
```

Есть шаблоны правил, например:

```
%.o : %.c %.h
    gcc -c -o $@ $<
```

`$@` — левая часть правила, `$<` — первая часть правой часть, `$^` — вся правая часть.

А ещё есть переменные. Например, каноническая переменная `CC = gcc` — компилятор C. Ещё канон — `CFLAGS`, там обычно указывают параметры компилятора. Использование через `$(CC)`. В круглых скобках указываются переменные Make-файла, в фигурных — shell'a.

Можно определить переменную `OBJS = $(patsubst %.c, %.0, $(wildcard *.c))`. `wildcard` пробежится по каталогу, собрав все .c файлы в список через пробел. `patsubst` все штуки вида `%.c` на `%.o`. И теперь, можно сделать `prog : $(OBJS)`

0.4 Работа с каталогами

Для работы с каталогами в C нужно подключить `<dirent.h>`. Есть `DIR`, по аналогии с соответствующим объектом для работы с файлами. Есть функции `opendir, readdir`, `closedir`ю

```
int stat(char*, struct stat*)
int lstat(char*, struct stat*)
int fstat(int fd, struct stat*)
```

Эти функции добывают разнообразную информацию о файле, записываемую в `stat`. `fstat` работает с уже открытым файлом. `lstat`, в отличие от `stat` работает ещё и с символическими ссылками. Символическая ссылка создаётся при помощи `link`, уничтожается при помощи `unlink`.

0.5 Низкоуровневый ввод-вывод

В ядре ОС есть две таблицы: таблица открытых файлов (ТОФ) и таблица открытых файлов для каждого процесса, в которой записями являются ссылки на записи ТОФ.

В каждой записи ТОФ хранится

- pid — список идентификаторов процессов
- файл устройства
- индексный дескриптор
- ссылка на очередь операций
- «курсор» в файле

Добавление записи в эту таблицу можно реализовать лишь системным вызовом

```
int open(char* path, mode_t mode, perm_t perms);
```

Возвращаемое значение — номер строчки в ТОФ. Для вывода ТОФ существует утилита `lsof`

Для манипуляций с ТОФ конкретного процесса есть следующий список системных вызовов:

```
int dup(int fd); //скопировать запись, возвращает номер строки, -1 при ошибке
int dup2(int oldfd, int newfd); //если не смог, -1, иначе 0
```

`dup2` принимает номера записей в ТОФ процесса, записывая в запись с номером `oldfd` содержимое записи с номером `newfd`.

По умолчанию у каждого процесса есть три записи:

- 0 — стандартный ввод
- 1 — стандартный вывод
- 2 — стандартный поток ошибок

В качестве прав передаются восьмеричные маски. Для `mode` есть следующие опции, комбинируемые побитовым ИЛИ:

- `O_CREAT` — создать файл, замяв существующие
- `O_EXCL` — не трогать существующие
- `O_APPEND` — добавление в конец файла
- `O_RDONLY`
- `O_WRONLY`
- `O_RDWR`
- Далее магия...
- `O_LARGEFILE` — использование 64-бытных смещений а 32-битной машине
- `O_CLOEXEC` — при смене тела процесса файл будет закрыт

- `O_SYNC` — все операции синхронизируются с драйвером внешнего устройства
- `O_DIRECTORY` — для открытия каталога

С открытым файлом можно использовать `fcntl` — изменение режима работы с файлом. Можно также заблокировать некоторый диапазон в файле. `flock` — выставление блокировки на файл целиком.

Также есть `ioctl` — выолняет... всякое разное с файловым дескриптором. Использование выглядит как

```
ioctl(int fd, int cmd, void*);
```

где `void*` — способ передачи или наоборот, получения каких-либо значений.

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

Эти функции принимают файловый дескриптор, буфер (в/из которого писать/читать) и количество нужных байт. В случае ошибки возвращают -1 и выставляют переменную `errno`. Иначе возвращают количество успешно прочитанных байт.

0.6 Порождение процессов

0.6.1 Создание процессов

`pid_t fork()` — породит процесс, возвращает идентификатор этого процесса.
`pid_t getppid()` — возвращает идентификатор родительского процесса.
`pid_t getpid()` — возвращает идентификатор самого процесса.
При исполнении `fork` будет создан точно такой же процесс, что и наш, с тем же кодом и скопированными данными и стеком.

0.6.2 Остановка процессов

А как можно завершить процесс?
`abort()` — неприличный способ убицца ап стенку. Процесс аварийно завершается. Если получится, будет создан дамп процесса.
`_exit(int status)` — системный вызов, завершающий текущий процесс с возвратом `status`.
`exit(int status)` — помимо завершения процесса, закрывает стандартные потоки ввода, вывода и ошибок.

0.6.3 Получение информации о процессе

`pid_t wait(int* status)` — блокировка процесса до остановки какого-то из дочерних процессов.

Возвращает идентификатор завершившегося процесса, записывая в `status` то, что было возвращено и немного служебной информации. Если на момент вызова `wait` дочерних процессов не было, то в `status` кладётся `-1` и процесс не блокируется.

`pid_t waitpid(pid_t pid, int mode, int* status)` — ожидание завершения конкретного процесса. В качестве режима можно указать, например,

- `WNOHUNG` — заставит процесс продолжить работу в случае, если `pid` живой.
- `WSTOPED` — проверка на то, остановлен ли процесс `pid`
- ...

Возвращает указанный `pid`, если всё корректно. Если в `pid` указан 0, то мы ждём любой дочерний процесс. Если указывать `-pid`, то будет ожидание группы с идентификатором `pid`.

А что делать с `status`? Для его разбора есть набор макросов:

- `WIFEXITED(status)` — вернёт `true` если при завершении процесса был вызван `_exit`, иначе `false`
- `WIFSIGNALED(status)` — если процесс был завершён передачей сигнала
- `WIFSTOPPED(status)` — если процесс был остановлен
- `WUNTRACED(status)` — если процесс не трассируемый, то есть не находится под отладчиком.
- `WEXITSTATUS(status)` — то, что было передано в `exit`, обрубленное до того, где не лежит служебная информация.

0.6.4 Замещение тела процесса

`execve(const char* path_to_file, char** args, char** env)` — замена тела процесса. Программа, на которую мы будем заменяться — первый аргумент. `args` — аргументы запуска этой программы, `env` — переменные окружения, причём переменные, которые были до этого, **не сохраняются**.

`execve` — системный вызов. Всё дальнеёшее — обёртки над ним, которые выглядят как `exec<suff>`. В качестве суффиксов есть следующие варианты:

- `e` — передача переменных окружения
- `p` — поиск программа в переменной `PATH`
- `v` — передача по `char**`
- `l` — передача через `(char* arg0, ...)`

1 Демонизация

В UNIX демон — это фоновый процесс, не взаимодействующий с терминалом и создающий новую сессию. Демоны запускаются процессом `init` с root-правами. Но это считается небезопасным, поэтому он должен сменить свои права на права другого, возможно, реального пользователя.

Есть команда `getpwuid`.

Сообщения о каких-то событиях записываются в специализированные файлы журнала. Потоки stdin и stdout обычно закрыты или перенаправлены.

Для смены корня можно использовать `chroot`.

Иногда нам хочется, чтобы одновременно был запущен лишь один демон соответствующего типа. Для этого обычно используется каталог `/var/run/`, в котором создаётся соответствующий классу демонов файл, который блокируется на время жизни демона.

По умолчанию считается, что демоны пишут свою информацию в syslog. Работа с ним начинается с вызова функции `openlog`. Для вывода используется

```
syslog(<log level>, <format string>, ...);
```

1.1 Как стать демоном?

У каждого процесса есть идентификатор сессии `sid` и идентификатор группы процессов `pgid`. Чтобы стать демоном нужно задать новый идентификатор процесса и новую процессную группу. И то, и другое можно создать только из собственного `pid'a`. но перед этим нужно закрыть все свои терминалы.

```
getsid
setsid
getpgid
setpgid
```

Вообще у каждого процесса есть

```
pid
ppid
sid
pgid
uid
gid
euid --- effective
egid --- effective
```

Но вернёмся к нашим баранам, а точнее — демонам. Как мы можем создать демона?

1. Закрыть стандартные потоки 0-2 при помощи `close`.

2. `fork`'нуться. А лучше дважды, чтобы исходное окружение нас вообще не трогало. Ну да ладно, один раз.
3. Меняем процессную группу.
4. Меняем сессию.
5. Меняем как надо `uid, gid, euid, egid`. Ведь обычно мы запускаемся `init`'ом, с root-правами, а это опасно.

2 Работа с файловой системой

Рассмотрим стандартные утилиты. В целом, содержание пакета `linux-utils`.

- `ls, dir`
- `touch`
- `find, locate`
- `mkdir, mv, cd, rm, rmdir, install`
- `chmod6 chown` — смена прав и владельца
- `stat`

  `chmod -R g-rwX, o-rwx`

  Здесь у группы отбираются права на чтение, запись и исполнение каталогов (нельзя в них сделать `cd`), а у остальных отбираются все права.

  Также на каталоге можно выставить бит `t` — travercy bit, и два бита `s` — то, от чьего имени запускать приложение

2.1 Работа с метаинформацией

Сюда входят утилиты п

В UNIX принятно, что файловая система создаётся при помощи

`mkfs.<system_name> <dev_name>`

Изменять параметры файловой системы можно, например, при помощи `tune2fs` — меняет метку (используется с файловыми системами `ext`)

`fsck` используется для проверки целостности файловой системы. `mount, umount` для монтирования и размонтирования системы соответственное.

2.2 Работа с файлами устройств для внешнего хранения

```
dd if=/devcdrom of=/archive/cd/music.iso
```

Эта команда просто, тупо, копирует побайтово копирует из `if` в `of`. Но надо быть осторожным. Даже осторожнее, чем с `reinterpret_cast` в плюсах.

Таблицы разделов

Есть два типа таблиц разделов — GPT и MBR. На самом деле, есть ещё Sun, SGI, BSD...

А ещё есть два семейства программ:

- `sfdisk, fdisk, cfdisk` — манипуляция с таблицами разделов MBR. `sfdisk` — абсолютно неинтерактивная манипуляция для использования в скриптах и простого переноса таблицы разделов с одного диска на другой. `fdisk` — обычная диалоговая система. `cfdisk` — более-менее приятный интерфейс.
- `sgdisk, gdisk, cgdisk` — то же самое, но для GPT.

2.3 Остальное

Есть `du` — disk-usage, `df` — disk-free.

3 Работа с потоком в терминале

- `more` — просмотр файлов.
- `less` — более умный просмотр файлов с построковой прокруткой, прокруткой влево-вправо и поиском при помощи `/`
- `head` — вывод начала файла
- `tail` — вывод конца файла.
- `cat` — КРАЙНЕ минималистичный редактор))
- `yes`
- `tee` — читать из стандартного ввода и выводить в стандартный вывод и в указанные файлы.
- `xargs` — может многое. Формирует из строки команду, запуская её.
- `cut` — вырезать из файла. Изменённый файл выплёвывается в стандартный поток вывода.

Базовые утилиты на этом заканчиваются.

- `sed` — строковый редактор, оперирующий строками и регулярками.
- `grep` — фильтр. Выводит все строки, в которых встречается данная строка
- `awk` — текстовый процессор. Наиболее используемая фишка — напечатать нужный столбец.

4 Манипуляция с процессами

- `ps` — список процессов (по умолчанию запущенных из данного шелла). Если нужно список всех процессов, `ps aux`.
- `top` — список процессов, отсортированных по использованию ресурсов
- `htop` — ещё круче, но нужно скачивать
- `kill` — убить (послать сигнал SIGTERM)
- `killall` — убить ВСЕХ (с данным префиксом)
- `renice` — смена приоритета процесса.
- `chroot` — смена корня для процесса и всех его потомков.
- `exec` — выполнить

5 Вернёмся к нашим бара... С

5.1 Как поговорить через сеть?

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

Для общения по сети есть целая одна функция

```
int socket(int domain, int discipline, int protocol)
```

- `domain` — семейство протоколов
- `discipline`
  - `SOCK_STREAM` — передача потока байт
  - `SOCK_DGRAM` — обмен сообщениями конкретного размера.
- `protocol` — протокол
  - `PF_INET`, `PF_INET6` — IPv4 И IPv6
  - `PF_DTM`
  - `PF_APPLETALK`
  - `PF_IPX`
  - `PF_UNIX` — универсальная заглушка. Можно использовать функции любого протокола, но взаимодействие возможно только в локальной системе и адресация происходит через файл в файловой системе.
  - `PF_RAW` — передача байт как есть
  - ...

Теперь будем разбирать механизмы `PF_INET`. Стоит почитать `man 7 ip`.

Протокол TCP — протокол с установлением соединения. Предполагается, что есть клиент и сервер.

Со стороны сервера происходит следующее:

- `id = socket`
- `bind(id)` — привязка к IP-адресу и порту

  порт — число типа `unsigned short`, причем 1-1024 зарезервированы ядром.
- `listen(id, 5)` — перевод сокета в слушающий режим, число — количество клиентов, которые могут одновременно находиться в режиме установления соединения
- ```
  while(1)
  {
      int client_fd;
      client_fd = accept(id, &address);
      if(fork() == 0)
      {
          работа с клиентом
      }
      close(client_fd);
  }
  ```
- При завершении работы сервера нужно закрыть socket.

Со стороны клиента:

- `id = socket`
- `connect(id, &address)`
- Взаимодействие....
- `close(id)`

5.2 Засада с адресами

1. Сетевой порядок байт. Для работы с этим используется набор функция `htons`, `ntohs`, `htonl`, `ntohl`.
2. DNS — Domain Name Services. Смысл этой штуки в том, чтобы адресу сопоставить некоторую строку. С его помощью можно по строке узнать IP адрес и наоборот. Для работы с ним есть две функции: `getaddrinfo` и `gethostbyname`, `gethostbyaddr`
  ```

6 IPC — Inter Process Communication

IPC — взаимодействие неродственных процессов внутри одного экземпляра операционной системы.

Есть два способа:

- POSIX IPC — более современный и популярный

  Для использования нужно линковать программу с `-lrt -lpthread`.

- system V — более сложный для программирования, но предоставляет больше возможностей и работает несколько быстрее

Средства взаимодействия процессов:

1. Разделяемая память. Грубо говоря, в ядре выделяется часть памяти, после чего каждый желающий процесс присоединяет его к своему виртуальному адресному пространству.
2. Семафоры `man 7 sem_overview`
3. Очереди сообщений. Сообщения

   - Именованные. С точки зрения system V каждому объекту сопоставлен некоторый ключ. Соответственно, если два процесса вычислили одинаковый ключ, то они оба могут пользоваться этим IPC-объектом

     С точки зрения POSIX именование объектов происходит как в файловой системе в формате `/.../.../...`. Процессы могут общаться по этому объекту, если у них есть одна и та же строчка.
   - Неименованные — не могут быть средством общения, так как другой процесс просто не может узнать о их существовании.

По умолчанию все IPC-объекты живут до момента выключения операционной системы (за исключением неименованных).

6.1 POSIX

Системные вызовы для использования IPC-объектов в POSIX выглядят как

- `ля_open` — создать IPC объект
- `ля_close` — отсоединиться от IPC-объекта.
- `ля_unlink` — уничтожить IPC-объект

6.2 System V

Следующие вызовы позволяют создавать и подключаться к IPC-объектам.

- `semget`
- `shmget`
- `msgget`

Отправлять разные команды объектам можно при помощи `***ctl`. Уничтожение происходит при передаче IPC_RMID.

Конвертация пути в ключ объекта System V:

```
key_t ftok(char* path, char color)
```

6.2.1 Разделяемая память aka shared memory

```
int shm_get(key_t key, size_t size, int flags)
```

Во флагах может быть

- `IPC_CREAT` — создание
- `IPC_EXCL` — если объект существует, то его не надо создавать и подключаться, получить сообщение об ошибке

Права доступа `rwxrwxrwx`.

Для присоединения памяти к процессу используется

```
void* shmat(int shmid, void* addr, int flags)
```

При передаче второго адреса не NULL, то разделяемая память будет отображена на этот самый виртуальный адрес (но нет), иначе отображение произойдёт на первые неотображённые страницы виртуальной памяти.

Отключение происходит при помощи `shmdt`

6.2.2 Семафоры

```
int semget(key_t key, int nsems, int flags)
```

Принимает ключ, количество необходимых семафоров и флаги.

Использование происходит при помощи

```
struct sembuf
{
unsigned short sem_num;
short sem_op;
short sem_flg;
}
```

Эта структура задаёт операцию с семафорами. Именно массив этих структур передаётся в

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

Самым важным полем является `sem_op`, которое может быть 3-ч видов:

- 0 — процесс блокируется до момента, когда значение семафора будет равно 0
- < 0 — Если можно вычесть, то значение семафора уменьшается и процесс продолжает работу, иначе процесс блокируется до момента возможности провести уменьшение.
- > 0

Документирующие комментарии

Документирующие комментарии позволяют при помощи таких утилит, как doxygen.
Стандартный стиль комментариев, принятый Java выглядит как

```
/** ...
 *  ...
 *  ...
 .
 .
 .
 */
```

Могут начинаться с `/*\!`, `/*#` и другие.

Пишутся комментарии перед объявлением функции или переменной. Также можно писать всякие разные атрибуты вида `@author Vasya Pupkin`, `@param abs <description>`, `@return ...`

Также могут быть недокументирующие комментарии вида

```
/*
 *
 *
 *
 */
```

Такие комментарии просто помогают понять написанный код

6.2.3 Очереди сообщений

Для использования очереди сообщений в POSIX нужно линковать программу с параметром `-lzt`.

Есть мнение, что очереди сообщений в POSIX несколько проще, чем в systemV.

Суть очередей сообщений в том, что в памяти ядра есть какой-то набор сообщений, к которым процессы могут получать доступ.

POSIX

Считается, что все сообщения приоритезированные, то есть у каждого сообщения есть тип и приоритет.

При попытке программы получить сообщение из очереди, он блокируется до получения сообщения, а сообщение, в свою очередь, извлекается из очереди

systemV

При приёме сообщения можно указать в параметрах `msgrcv` (принять сообщение) type — тип принимаемого сообщения.

- 0 — получить сообщение из очереди
- >0 — получить сообщение нужного типа
- <0 — получить сообщение с наименьшим приоритетом, меньше модуля заданного значения (аналог приоритета)

Парной функцией к `msgrcv` является `msgsnd`. Сообщение с нулевым типом отправить нельзя

Сообщение представляет из себя некоторую структуру вида

```
struct msgbuf
{
long mtype;
char mtext[1];
}
```