1 Основные понятия.

Цели курса:

- узнать, для чего нужны операционные системы;
- познакомиться с краткой историей ОС;
- получить понятия об архитектуре ОС и её компонентах;
- изучить основы взаимодействии прикладных задач и ОС;
- изучить взаимодействия ОС с оборудованием;
- ознакомиться с понятием системного вызова и с некоторыми системными вызовами;
- научиться использовать системные вызовы для создания эффективных прикладных программ.

1.1 Что будем делать на семинарах

На семинарах мы будем:

- изучать архитектуру ОС на примере ОС Linux;
- писать небольшие программы, иллюстрирующие использование возможностей ОС;

1.2 Знакомство с Unix-подобными ОС

Unix-like OS — самые распространённые в мире ОС.

- 99.99% планшетов;
- 97% смартфонов;
- 95% кластеров и высокопроизводительных рабочих станций;
- 15% домашних компьютеров и ноутбуков;
- 70% роутеров;
- 95% систем хранения данных;
- + телевизоры, бортовые компьютеры автомобилей, ...

1.3 Отличие двух подходов, Unix и Windows

- Unix-like — небольшое исчерпывающее количество точек взаимодействия прикладных программ и ОС (несколько сотен *системных вызовов*). Создание новых средств не приводит к увеличению количества системных вызовов. Набор средств представляет базис (устоявшееся, но не совсем верное название — ортогональный набор).
- Windows-like — попытка создать новый набор системных вызовов для каждого нового средства. Сотни тысяч точек взаимодействия прикладных программ и ОС (`API`). В каждой новой версии появляются десятки тысяч новых API. Очень много устаревших API. Одно и то же действие можно сделать десятками различных способов (неортогональный набор API).

1.4 Вход в систему

Получить имя пользователя и пароль у администратора.

Все вводят свои данные.

При вводе пароля он может не отображаться!

Если мы попадаем в оконный режим, то находим в меню терминал в системных программах и в дальнейшем работаем в нём.

У всех должно появиться приглашение, заканчивающееся на `$`.

```
$
```

1.5 Первое задание: программа 1

Все должны написать программу на языке `C`, выводящую строку

```
Hi again.
```

Можно на бумаге или доске.

```
#include <stdio.h>
int main() {
  printf("Hi again\n");
  return 0;
}
```

Объяснение программы, если кто не помнит:

Программа всегда начинается с функции `main`.

Все функции, которые вызываются, должны быть описаны ранее. Мы используем функцию `printf`, которая описана во включаемом системно файле `stdio.h` (об этом говорят знаки `<` и `>`). Наши собственные включаемые файлы выглядят так:

```
#include "myfile.h"
```

1.6 Первая цель: программу нужно запустить.

Текст программы хранится в *файле*.

Что такое файл?

Файл — неупорядоченный набор байтов, один из объектов *файловой системы*.

Какие ещё объекты файловой системы мы знаем?

Директории, *каталоги*, *папки* — синонимы. В первом приближении директории — объекты, которые могут содержать в себе другие объекты.

Вернёмся к тому моменту, когда мы вошли в систему. Мы видим знак доллара. Это — приглашение системы у вводу команд или просто *приглашение* (*prompt*). Мы что-то вводим, система отвечает. Зачем такие сложности управления системой?

- Из командной строки можно сделать абсолютно всё, что умеет система.
- Все высокопроизводительные компьютеры и кластеры управляются именно из командной строки. Для работы с кластером знание командной строки обязательно!
- Мы быстро научимся писать полезные программы, используя командную строку. Если бы мы хотели использовать даже примитивнейший графический интерфейс, нам пришлось бы потратить пару месяцев.

1.6.1 Формат командной строки

Командную строку принимает к исполнению и анализирует *интерпретатор команд*, *шелл(shell)* или *оболочка*.

```
$abra -flag1 -flag2 cadabra
```

Командная строка разбивается интерпретатором на *аргументы*. Первый аргумент — всегда имя команды, в данном случае, `abra`. Аргументы, начинающиеся с минуса обычно называются *флагами* команды и меняют поведение программы (`flag1` и `flag2`). Остальные аргументы часто являются именами файлов, но это не обязательно.

```
$man abra
```

для системных программ обычно выведет на экран документацию по команде `abra`.
Можно поискать, в каких статьях справки содержится слово `copy`

```
$apropos copy
```

1.6.2 Структура файловой системы. Текущая директория

В первом приближении файловая система строго иерархическая. На верхнем уровне — *корень*, *root*, обозначающийся прямым слэшем `/`. Другое название — корневая директория. В ней находятся какие-то файлы и другие директории. Каждая из директорий может содержать новые файлы и директории. Обычно в корневой директории содержатся системные директории.

- `bin` — содержит файлы с исполнимыми программами системы
- `usr` — содержит системные компоненты системы
- `lib` — библиотеки системы. Например, там имеется библиотека `libc`, в которой содержится код и данные для языков `C` и `C++`.
- `home` — директории пользователей.
- кучу других системных директорий.

Короткое имя файла — уникально в директории. Не может содержать символов `'/'` и символа с кодом 0.

Полное имя файла — создаётся конкатенацией всех вышестоящих имён директорий с добавлением разделителей `'/'`. Например: `/usr/local/bin/co`

Текущая директория.

Получить имя текущей директории:

```
$pwd
```

Относительное имя файла — определяется от текущей директории. В любой директории всегда имеются директории `"."` (сама она) и `".. "` (вышестоящая в дереве, *родительская*). Например: `./../../../usr/sbin/netstat`

Домашняя директория — директория, являющаяся текущей после регистрации пользователя в системе.

Сменить текущую директорию:

```
cd name_of_directory
cd
```

Второй вариант сменит директорию на домашнюю.

1.7 Возвращаемся к написанию программы. Цикл разработки программы.

1.7.1 Создание или редактирование файла с текстом программы.

Кто чем умеет, тот тем и пользуется. Например, `vi h.c`.

Чуть-чуть о `vi`:

- Имеет два режима — командный и редактирования. Переход в командный режим — `ESC`, в режим редактирования из командного режима — `i`.
- в режиме редактирования работают стрелки, набираемый текст появляется в *буфере*.
- для сохранения файла переходим в командный режим по `ESC`, затем набираем `:wq`.
- если мы наделали чепухи, можно выйти из командного режима без сохранения по `:q!`

Набираем файл с исходным кодом, например с именем `h.c`

1.7.2 Немного работы с файлами

Посмотрим, существует ли файл.

```
$ls -l h.c
-rw-r--r--  1 student  students  71 Sep 3 17:00 h.c
$
```

`ls` — имя команды просмотра содержимого директорий (список объектов директории). Флаг t-lt выдаёт список в длинном формате. 71 — длина файла в байтах. `student` — имя пользователя. `students` — имя группы. Об этом чуть попозже.

Распечатаем файл на экране:

```
$ cat h.c
#include <stdio.h>
int main() {
  printf("Hi again\n");
  return 0;
}
$
```

Скопируем файл:

```
$cp h.c h_copy.c
$
```

Посмотрим на содержимое директории:

```
$ls -l
-rw-r--r--  1 student  students  71 Sep 3 17:00 h.c
-rw-r--r--  1 student  students  71 Sep 3 17:08 h_copy.c
$
```

Удалим один файл:

```
$rm h_copy.c
$
```

Посмотрим, что осталось:

```
$ls -l
-rw-r--r--  1 student  students  71 Sep 3 17:00 h.c
$
```

Кстати, команда `rm -rf /`, выданная соответствующим пользователем, удалит все доступные файлы, начиная с корневого каталога рекурсивно (`r`) и без подтверждения (`f`, от слова force).

1.7.3 Компиляция программы

Компилируем программу:

```
$cc h.c
$
```

Опять посмотрим, что в директории:

```
$ls -l
-rwxr-xr-x   1 student  students Sep 3 31 17:11 a.out
-rw-r--r--  1 student  students  71 Sep 3 17:00 h.c
$
```

Появился файл `a.out`. Запускаем его:

```
$./a.out
Hi again
$
```

Первая программа под Unix готова!

1.7.4 Что происходило при компиляции

При вызове команды компиляции происходило следующее:

- Компиляция программы `h.c` в объектный файл `h.o`, содержащий машинные команды нашего процессора и ссылки на библиотечные функции (*неразрешённые внешние ссылки*). В нашем файле нет кода для функции `printf`, она и есть одна из неразрешённых внешних ссылок.
- Нахождение кода для всех неразрешённых внешних ссылок в библиотеке `libc` и объединение (*линковка*, *сборка*) всего в единый исполнимый файл с именем `a.out`.

Задание: Сделайте копию файла `h.c`, измените в файле вызов функции `printf` на `myprintf`. Скомпилируйте. Посмотрите результат.

2 Знакомство с системными вызовами.

2.1 Пользователи и группы

Вспомним, что у нас находится в директории:

```
$ls -l
-rwxr-xr-x  1 student  students Sep 3 31 17:11 a.out
-rw-r--r--  1 student  students  71 Sep 3 17:00 h.c
$
```

Пропустим пока первые две позиции. В третьей и четвёртой позициях мы видим имя пользователя и имя группы, которые владеют этим файлом. Для данных файлов оно совпадает с тем именем, которое дал администратор. Для чужих файлов результат будет другим. Например:

```
$ls -l /bin/c*
-rwxr-xr-x  1 root  wheel   19648 Dec 19  2013 /bin/cat
-rwxr-xr-x  1 root  wheel   26080 Dec 19  2013 /bin/chmod
-rwxr-xr-x  1 root  wheel   24848 Dec 19  2013 /bin/cp
-rwxr-xr-x  2 root  wheel  357984 Dec 19  2013 /bin/csh
$
```

Мы воспользовались *маской* имени файла, используя символ `'*'` . Этот символ служит для сопоставления имени файла образцу и относится к *метасимволам*. Он может сопоставляться с любым, в том числе и нулевым, количеством любых символов. Другой метасимвол `'?'` сопоставляется ровно с одним символом имени. Например,

```
$ls -l /bin/c??
-rwxr-xr-x  1 root  wheel   19648 Dec 19  2013 /bin/cat
-rwxr-xr-x  2 root  wheel  357984 Dec 19  2013 /bin/csh
$
```

Кто занимается сопоставлением? Интерпретатор команд. Это он выбирает имена файлов, соответствующие образцу и производит так называемое *расширение имён*, заменяя аргумент, содержащий метасимволы, именами файлов, удовлетворяющими условию сопоставления. Если таких имён файлов нет, аргумент с метасимволами остаётся неизменным.

Если вы создадите файл, содержащий метасимволы, с ним будет трудно работать в оболочке, но можно. Каждый метасимвол может быть заэкранирован бэкслешем, например, `a\*b` это именно файл с таким именем, а не файл, начинающийся с `a` и кончающийся `b`.

Вернёмся к пользователям. Каждый пользователь состоит по крайней мере в одной группе (может и в нескольких). Права пользователя на файл определяются так называемыми *битами защиты* файла[1].

Первый символ в выводе команды `ls` в каждой строке показывает тип объекта файловой системы (пока мы знаем только простые файлы, *plain files*, для которых первым символом будет `'-'` , и директории, показываемые символом `'d'`).

Следующие девять символов есть биты защиты файла. Прочерк означает отсутствие прав, буква — наличие прав. Эти девять битов разбиты на три группы по три бита в каждой. Эти группы битов называем **я**, **группа**, **остальные**. Значения битов немного зависят от типа объекта. Например, для файла первый бит группы — `'r'` — означает право чтения файла. Второй — `'w'` — право записи в файл. Третий — `'x'` — право исполнения файла (об этом чуть дальше). Для директорий `'r'` есть право получения списка файлов в директории, `'w'` — право изменения директории, например, создание и удаление в ней файлов. `'x'` означает право прохождения через директорию, то есть, право назначать директорию текущей а так же право получения дополнительной информации о файлах в этой директории[2].

Удобно представлять эти биты в восьмеричной системе, младший бит справа. Полные права доступа тогда будут представляться тремя восьмеричными цифрами, равными сумме входящих в них битов. Бит `'r'` есть 04, бит `'w'` есть 02, бит `'x'` есть 01. Тогда комбинация `rwxr-xr-x` представится как `0755`.

Каким образом в системе представляются пользователи и группы? Мы видим, что имя пользователя наших файлов — `student`. Значит ли это, что при запросе пользователя у системы мы получим строчку `"student"`?

Нет. В операционной системе и пользователи и группы представляются не текстовыми строчками, а числами. Давайте напишем небольшую программу по определению числового идентификатора текущего пользователя.

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main() {
    uid_t uid = getuid();
    gid_t gid = getgid();
    printf("UID=%ld, GID=%ld\n", (long)uid, (long)gid);
    return 0;
}
```

В нашей программе много нового и пока непонятного. Во-первых, появились два новых включаемых файла. Во-вторых, появились незнакомые идентификаторы `uid_t` и `gid_t`. В-третьих, появились новые незнакомые функции `getuid` и `getgid`.

Новый включаемый файл `<sys/types.h>` потребовался для описания того, каким именно образом представляются числовые идентификаторы пользователя и группы. Этот файл

[1] В современных версиях Unix-like систем имеются и более сложные механизмы защиты, но базовый механизм, основанный на битах доступа, имеется во всех.

[2] Если трактовать директорию как объект, содержащий другие объекты, то всё становится на свои места — `'r'` есть право чтения имён объектов, `'w'` — право изменять список объектов.

содержит описания новых (для нас) типов данных `uid_t` и `gid_t`, которые расширят доступную нам систему типов языка. Второй включаемый файл `<unistd.h>` содержит описание *системных вызовов* `getuid`, который возвращает нам числовой идентификатор пользователя и `getgid`, который возвращает нам числовой идентификатор группы. Эти два файла будут преследовать нас весь семестр. Потом добавятся ещё несколько...

Конструкция `(long)uid` есть преобразование переменной `uid`, которая, как мы говорили, есть какой-то целочисленный числовой тип к типу `long`, который, как мы знаем, выводится по формату `%ld`. Этот механизм называется преобразование типов или по-английски *cast* или *casting*. В `C++` есть ряд ключевых слов, содержащих суффикс `cast`, например, `reinterpret_cast`.

Для понимания операционных систем очень важно понимать, что программы сами по себе не создаются, не запускаются, не выполняются и не заканчиваются. Программы проявляют активность в виде *процессов*, объектов операционной системы. Процессы не могут взаимодействовать с пользователем, аппаратурой и другими процессами напрямую. Для исполнения какого-то подобного действия, процесс должен выдать операционной системе запрос, который та может выполнить, а может отказать в выполнении. Так как язык `C` является главным языком в UNIX, языком реализации системы, то такие запросы оформляются в виде функций языка `C`. Реализация этих системных вызовов, конечно, зависит от архитектуры вычислительной системы и она обычно состоит из нескольких специализированных машинных команд. Исполнение этих команд переводит исполняемый процесс в другой режим работы (режим ядра в отличие от пользовательского режима). В этом режиме процессору (не процессу!) доступны любые машинные команды, в том числе команды взаимодействия с оборудованием, недоступные из режима пользователя. Конечно, при исполнении системных вызовов тщательно проверяются все полномочия выдавшего из процесса.

При системных вызовах исполняется код ядра ОС, при вызове обычных функций исполняется код процесса, часть из которого написана нами, часть добавляется из библиотек.

Как просто определить, какому номеру пользователя соответствует какое имя? Можно запустить команду

```
$grep root /etc/passwd
root:*:0:0:System Administrator:/var/root:/bin/sh
$
```

Пользователь `root` как видно имеет номер `0`, номер группы `0`, домашнюю директорию `/var/root` и назначенный интерпретатор команд, шелл, `/bin/sh`.

Создать пустой файл или изменить время существующего файла на текущее можно полезной программой `touch`:

```
$touch foo
```

Посмотрев на него мы увидим, что его биты защиты имеют какое-то значение по умолчанию, обычно `0644`. Стандартно при создании файла ему даются биты защиты `0666`, то есть чтение/запись всем, а исполнимым — `0777`, добавляя бит исполнения для всех. Почему у нас получилось `0644`?

Наберём

```
$umask
0022
$
```

Мы увидели маску для создания файлов в текущем процессе. Для понимания, как она работает, надо понимать битовые операции. При **создании** файла (и именно её!) назначенные нами биты сбрасываются в ноль для битов, установленных в `umask`.

```
0110110110 --- назначенные (0666)
0000010010 ---  umask              (0022)
0110100100 --- результат        (0644)
```

Это не операция вычитания!!! Например, для назначенных битов `0500` и маске `0022` результат будет `0500`!

Посмотрим, как мы можем изменить биты защиты файла. Для этого есть системный вызов `chmod` а так же программа `chmod`. Запросив справку по `chmod` мы получим описание программы.

```
$man chmod
```

Посмотрев справку, мы увидим, что простейший способ изменить биты защиты файла будет таким:

```
$chmod 0711 foo
$ls -l foo (убеждаемся)
$chmod 0600 foo
```

Другие способы смотрите в `man`

Теперь мы хотим изменить биты защиты именно в нашей программе. Для этого потребуется системный вызов `chmod`. Как же получить по нему справку?

```
$man 2 chmod$
```

Мануал содержит несколько разделов. Раздел `1` — команды системы. Раздел `2` — системные вызовы. Раздел `3` — библиотечные функции. Раздел `4` — структуры данных и т. д.

Файл `h2.c`:

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main() {
  int err = chmod("foo", 0711);
  printf("error=%d\n", err);
  return 0;
}
```

Компилируем и запускаем:

```
$cc h2.c
$./a.out
error=0
$ls -l foo
$rm foo
$./a.out
error=-1
$
```

Системный вызов `chmod`, в отличии от системных вызовов `getuid` и `getgid` возвращает индикатор успеха. Часто возвращённый нуль означает, что всё хорошо, минус единица, что была ошибка при исполнении системного вызова. Что возвращает конкретный системный вызов обязательно смотреть в справке!

Как узнать причину ошибки?

Используем включаемый файл `errno.h`

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
int main() {
  int err = chmod("foo", 0711);
  printf("error=%d, errno=%d\n", err, errno);
  return 0;
}
```

Опять компилируем и запускаем:

```
$cc h2.c
$./a.out
error=-1 errno=2
$
```

Вроде лучше. Ещё лучше так:

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
int main() {
  int err = chmod("foo", 0711);
  perror("foo");
  return 0;
}
```

```
$cc h2.c
$./a.out
foo: No such file or directory
$
```

Уже вразумительно.
Заменим `"foo"` на `"/bin/cp"` .

```
$cc h2.c
$./a.out
/bin/cp: Operation not permitted
$
```

При написании проектов использование обработок ошибок обязательно!
Вопросы:

1. Является ли `printf` системным вызовом?
   Нет. Это — функция из библиотеки `libc`.
2. Как получить справку по системному вызову `stat`?
   `man 2 stat`

Ещё вопросы из книги.

3 Процессы.

3.1 Компоненты процесса

Итак, процесс есть воплощение программы. Программа существует в виде исполнимого файла, процесс — исполняющаяся программа.

Контекст процесса — то, чем один процесс отличается от другого. Вся деятельность процесса — работа по изменению его контекста, Делим на

- пользовательский контекст
- системный контекст

Пользовательский контекст:

- код процесса (код функции `main` и всех вызываемых функций) (`TEXT`). В скобках — принятые названия секций процесса.
- данные процесса (все локальные и глобальные переменные, пользовательские и из подключаемых библиотек). Они тоже делятся на группы:

- инициализируемые статические данные (DATA)
- неинициализируемые статические данные (BSS)
- неизменяемые статические данные, константы (CONST)
- стек (STACK)
- динамически заказываемая память по malloc, calloc, new (*куча*) (HEAP)

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

int gx = 5; // DATA
int nx;       // BSS
const int PI = 3.1415926; // CONST

int main() {
   int lx; // STACK
   int ly = 10; // STACK
   static int sx; // BSS
   static int isx = 123; // DATA
   const int cx = 123; // CONST
   int *hx; // STACK  // *hx - мусор (garbage)
   hx = malloc(100); // *hx --> HEAP
   free(hx); // *hs --> висячая ссылка (hanging reference)
}
```

В исполнимом файле содержатся:

- TEXT
- CONST
- DATA

BSS заполняется нулями при загрузке процесса в память. STACK и HEAP выделяется при работе.

Команда size.

```
size a.out
TEXT DATA BSS TOTAL
2304  154    700

int zzz[100000];
int main() {}

$cc a.c
$size a.out

int zzz[100000] = {123};
int main() {}

$cc a.c
$size a.out

int main() {
  int zzz[100000];
}

$cc a.c
$size a.out
```

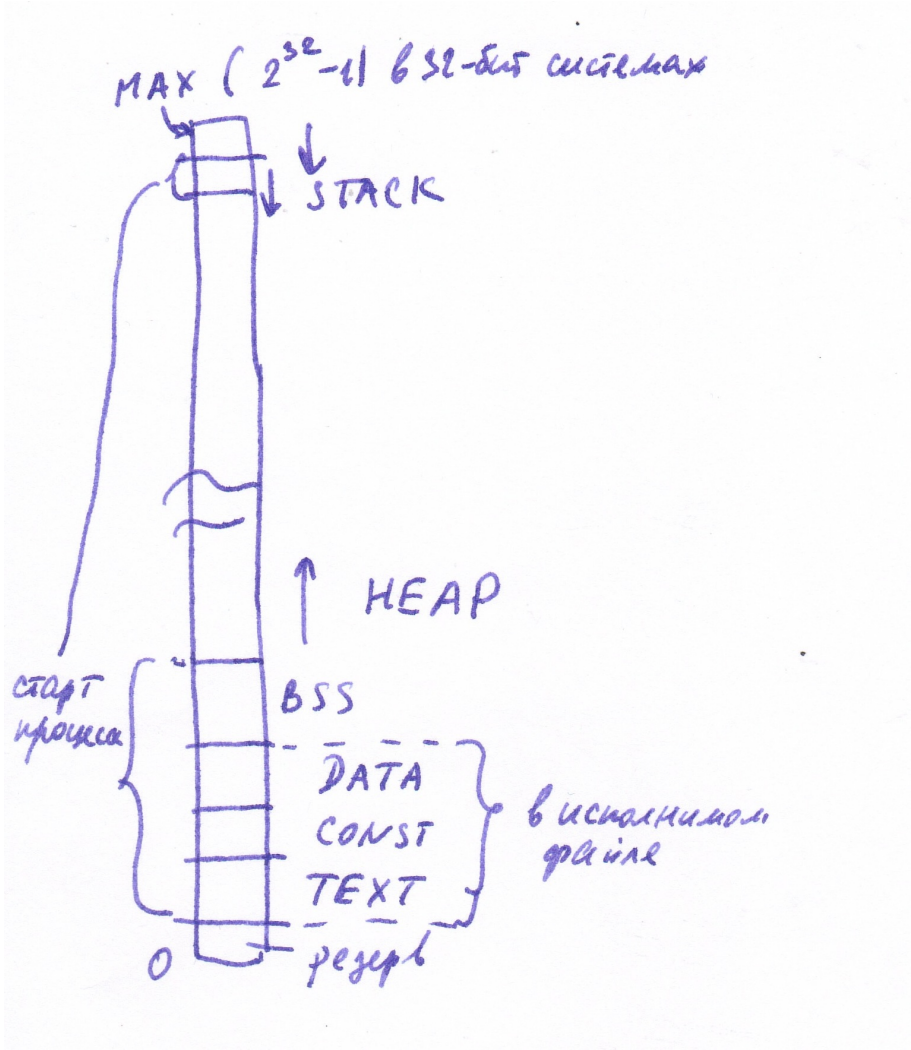

Рис. 1: Распределение памяти процесса

```
int main() {
  int zzz[100000];
  printf("hello\n");
}

$cc  a.c
$size a.out
```

Другая часть контекста — системная. Хранит

- стек пользователя в контексте ядра (для системных вызовов)
- данные ядра (пока мы знаем про идентификатор пользователя процесса и его группы). `Process Control Block`

Получить идентификатор самого процесса:

```
// h22.c
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
```

```
int main() {
   pid_t pid = getpid();
   pid_t ppid = getppid();
   printf("PID=%ld, PPID=%ld\n", (long)pid, (long)ppid);
  return 0;
}

$cc h22.c
$./a.out
PID=1534 PPID=154
$a.out
PID=1537 PPID=154
$
```

Все процессы исполняются в дереве процессов. Прародитель всех (Адам) — процесс `init`. Вопрос:

- Почему `PPID` одинаков?

  Потому, что это `PID` общего родителя — оболочки

3.2 Порождение процессов. `fork`

.

Породить новый процесс в UNIX можно ровно одним путём — через системный вызов `fork()`.

Добавим в предыдущую программу одну строку.

```
// h23.c
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main() {
   fork();
   pid_t pid = getpid();
   pid_t ppid = getppid();
   printf("PID=%ld, PPID=%ld\n", (long)pid, (long)ppid);
  return 0;
}

$cc h23.c
$a.out
PID=78190, PPID=42144
PID=78191, PPID=78190
$
```

OOPS. Мы написали программу, выводящую одну строку. Мы её запустили. Получили две строки на выходе. Как назвать то, что произошло?

- Мы запустили две программы?
- Мы запустили два процесса?
- Мы запустили программу, которая вывела две строки?
- Мы запустили программу, которая в своём процессе создала новый?

Правилен последний ответ. Наша программа была запущена кем? Оболочкой. Наша программа системным вызовом `fork()` породила ещё один процесс, который и вывел строку.

А что за новый процесс? Что он исполняет? Он исполняет точно ту же программу, из процесса которой он вызван. `fork()` создаёт точную копию вызывающего процесса с незначительными отличиями:

- Разные идентификаторы процессов.
- Системный вызов `fork` возвращает разное значение для родителя и ребёнка

```
// h24.c
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main() {
   pid_t pid = fork();
   if (pid == -1) {
      perror("fork failed");
   } else if (pid == 0) {
     printf("Hi, I am the child, my pid is %ld\n", (long)getpid());
   } else {
     printf("Hi, I am the parent, my child pid is %ld\n", pid);
   }
   return 0;
}

$cc h24.c
$a.out
Hi, I am the parent, my child pid is 78318
Hi, I am the child, my pid is 78318
$
```

С точки зрения операционной системы при `fork()` создаётся новая запись в таблице процессов ядра операционной системы, создаётся новый системный контекст порождённого процесса и создаётся копия пользовательской части порождённого процесса.

Одна программа — два независимых процесса. Шизофрения.

Для продвинутых можно чуть забежать вперёд и рассказать про то, что на самом деле пользовательская часть не копируется, а расслаивается (`COPY-ON-WRITE`).

3.3 Интересные особенности `fork`.

Сколько звёздочек будет напечатано при исполнении следующей программы? Почему? (15)

```
// h25.cc
#include <unistd.h>
#include <stdio.h>
int main() {
  for (int i = 0; i < 4; i++) {
    printf("*\n"); fork();
  }
  return 0;
}

$c++ h25.cc
$a.out
```

А в этой? Почему? (30)

```
// h26.cc
#include <unistd.h>
#include <stdio.h>
int main() {
  for (int i = 0; i < 4; i++) {
    fork(); printf("*\n");
```

```
  }
  return 0;
}
```

Что выведет следующая программа:

```
//h27.cc
#include <unistd.h>
#include <stdio.h>
int main() {
  printf("Hi");
  fork();
}

$c++ h27.cc
$a.out
HiHi$
```

Объясните результат.

А эта?

```
//h28.cc
#include <unistd.h>
#include <stdio.h>
int main() {
  printf("Hi\n");
  fork();
}

$c++ h28.cc
$a.out
Hi
$
```

3.4 Завершение процесса. Функция (не системный вызов!) `exit` и системный вызов `_exit`. Семейство `wait`

Процесс завершается:

- При исполнении системного вызова `_exit`. Процесс немедленно завершается (переводится в состояние *закончил исполнение*).
- При исполнении функции `exit`. Сбрасываются все буфера ввода/вывода и после этого вызывается `_exit`.
- При завершении функции `main` любым способом — через `return` или при достижении завершающей фигурной скобки. Аналогично `exit`.

То есть, всё всегда заканчивается `_exit`.

Если `_exit` декларируется в `unistd.h`, то `exit` — в `stdlib.h`.

Аргумент `_exit` можно получить в родительском процессе системным вызовом семейства `wait` (декларируется в `sys/wait.h`).

Простейший вызов `wait`:

```
// h28.cc
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
```

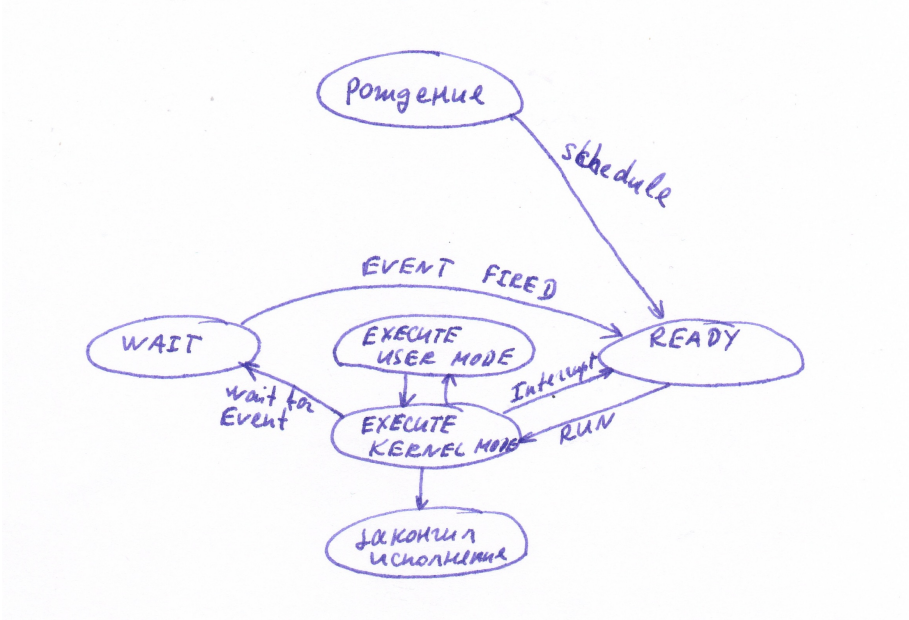

Рис. 2: Жизнь процесса

```
int main() {
  pid_t pid = fork();
  if (pid == -1) {
    perror("Fork failed");
  } else if (pid == 0) {
    printf("Hi, child about exit(10)\n");
    exit(7);
  } else {
    printf("Waiting for child...\n");
    int code;
    wait(&code);
    printf("Waiting done with code %d\n", code);
  }
  return 0;
}
```

```
$c++ h28.cc
$a.out
Hi, child about exit(10)
Waiting done with code 2560
$
```

Код возврата, аргумент `exit(10)` находится в битах с 9 по 16-й переменной, полученной системным вызовом `wait`.

Соглашение по кодам возврата из процесса следующее:

- Если процесс завершился успешно, то код возврата должен быть равен 0 (`exit(0)`).
- Если процесс завершился неуспешно, то код возврата ненулевой. Какой именно — если это важно, то это документируется в справке по программе.

Код возврата может использоваться в командных файлах. В частности, следующая программа может запускаться при успешном завершении предыдущей (или при неуспешном!).

```
$ ls -l qwerty || echo "no querty"
```

```
ls: qwerty: No such file or directory
no querty
$ ls -l h28.cc && echo "h28.cc exists"
-rw-r--r--  1 student  student  412 Sep  7 17:41 h28.cc
h28.cc exists
$
```

3.5 Аргументы функции `main`

Если с возвращаемым значением функции `main` мы разобрались — это аргумент для заключительного вызова функции `exit`, то аргументами функции `main` мы ещё не пользовались.

Полный синтаксис функции `main` следующий:

```
int main(int argc, char **argv, char **envp) {
  ...
}
```

Заметим, что по синтаксису языка `C` второй и третий аргумент можно записывать и в другой форме:

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

Несмотря на синтаксическую сложность записи, смысл этих аргументов прост — эти аргументы можно рассматривать как массив строк языка `C`. Первый аргумент, который обычно называют `argc` – **arg**uments **c**ount – есть размер массива `argv` – **arg**uments **v**ector – содержащего эти строки. Имена `argc` и `argv` не предопределены и не зарезервированы, можно использовать любые имена переменных, важен только тип. Эти имена обычно применяются для удобства и для уменьшения количества комментариев в программе :).

Третий аргумент — среда процесса, массив строк, содержащий так называемые переменные среды. Если для первого массива строк, указывается точная длина, `argc`, то для массива `envp` она не указывается. Сам массив `envp` содержит в себе ограничитель — его последний элемент есть указатель `NULL`. За ним ничего нет! Попытка обратиться к элементу массивы `envp` за элементом, содержащим `NULL` скорее всего приведёт программу к краху.

Задачи:

1. Напишите программу, которая распечатывает свои аргументы через пробел.

   ```
   #include <stdio.h>
   int main(int argc, char **argv) {
       for (int i = 1; i < argc; i++) {
           printf("%s ", argv[i]);
       }
   }
   ```

2. Напишите программу, которая распечатывает все переменные среды.

   ```
   #include <stdio.h>
   int main(int argc, char **argv, char **envp) {
       for (int i = 0; envp[i] != NULL; i++) {
           printf("%s ", envp[i]);
       }
   }
   ```

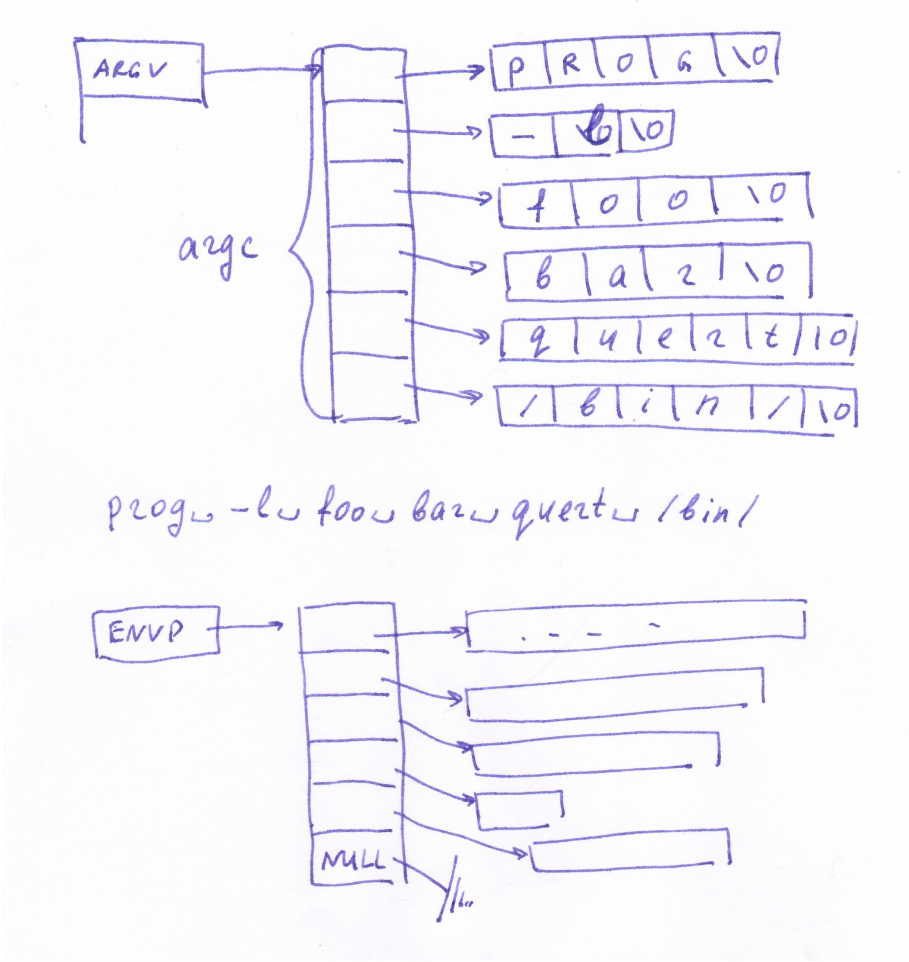

Рис. 3: Аргументы main

3.6 Изменение пользовательского контекста процесса. Группа функций `exec`.

Если единственным способом создать новый процесс является выдача системного вызова `fork`, то единственным способом исполнить какую-либо программу является выдача системного вызова `exec`. Сам системный вызов `exec` обычно явно не вызывают, его аргументы достаточно сложны и не всегда удобны, вместо этого пользуются одной из более удобных функций.

```
//h209.c
#include <unistd.h>
int main() {
  execl("/bin/ls", "ls", "-l", "/", NULL);
}
```

Откомпилируйте и выполните эту программу.

Первый аргумент `execl` — имя исполнимого файла, образ которого загружается в пользовательский контекст процесса. Он полностью заменяет все рассмотренные секции памяти (виртуальной памяти, об этом попозже) содержимым из указанного исполнимого файла. Управление передаётся, как и во всех программах, на функцию `main`. Аргументами этой функции становятся строки, переданные в параметрах вызова `execl`, начиная со второго. То есть, начнёт исполняться программа, образ которой находится в файле с именем `"/bin/ls"` с аргументами

```
argc = 3
```

```
argv[0] = "ls"
argv[1] = "-"
argv[2] = "/"
```

Если функция завершилась успешно, то больше ничего в текущем процессе выполняться не будет! Можно сказать, что это — функция без возврата. При неуспешном завершении выполнится, как обычно, следующая за функцией строка.

Аргумент **envp**, среда процесса, будет передана неизменной.

При запуске команд мы обычно не задумываемся о том, является ли команда встроенной командой оболочки или внешней программой. Обычно оболочка позволяет определить, чем является команда. Например, оболочка `bash` имеет встроенную команду `type`.

```
$type c++
c++ is hashed (/usr/bin/c++)
$ type cd
cd is a shell builtin
$ type type
type is a shell builtin
$
```

Модификация функции под именем `execlp` позволяет выполнить программу, не зная заранее полное имя её исполнимого файла. Когда мы распечатывали переменные среды `envp`, мы могли заметить переменную по имени `PATH` (хотя как её заметить, если не знать, что искать?). В этой переменной хранится список директорий, в которых ищутся исполнимые программы. Поиск завершается при нахождении исполнимого файла в одной из указанных в этой переменной среды директорий.

Если мы не знаем, где находится, например, исполнимый файл программы `ls`, то лучше воспользоваться именно этим вариантом функции:

```
// h210.cc
#include <unistd.h>
int main() {
  execlp("ls", "ls", "-l", "/", NULL);
}
```

Итак, суффикс `"p"` означает *path*.

Суффикс `"l"` означает *list*, то есть, аргументы передаются списком, заканчивающимся `NULL`. Честно говоря, сценарии использования именно формы со списком ограничены случаями, когда заранее точно известно число аргументов. Чаще число аргументов определяется лишь при исполнении программы. В этом случае требуется другой вариант функции, с суффиксом `"v"` от слова *vector*.

```
//h211.cc
#include <unistd.h>
int main() {
  char *args[100]; // Массив указателей на 100 указателей.
  args[0] = "ls";
  args[1] = "-l";
  args[2] = "/";
  args[3] = NULL;
  execv("/bin/ls", args);
}
```

Не очень хорошая практика резервировать место на 100 указателей — на сколько бы указателей мы бы не зарезервировали место, всегда найдётся программа, которая этот резерв преодолеет. В реальной жизни требуется аккуратное использование функций по заказу памяти из кучи таких, как `calloc`, `malloc`, `realloc` и, конечно, `free`.

```
//h212.cc
```

```
#include <unistd.h>
int main() {
  char *args[100]; // Массив указателей на 100 указателей.
  args[0] = "ls";
  args[1] = "-l";
  args[2] = "/";
  args[3] = NULL;
  execv("/bin/ls", args);
}
```

Следующая программа исполняет программу, имя которой и аргументы вводятся с клавиатуры по аргументу на строку. Пустая строка обозначает конец ввода. Имя программы и каждый из аргументов не может быть длиннее 100 символов, самих аргументов тоже не может быть больше 100 символов. Эта программа интересна лишь с точки зрения иллюстрации особенности функции `execv`, к тому же она использует запрещённую ныне функцию `gets`.

```
//h213.cc
#include <unistd.h>
#include <stdio.h>
#include <string>
int main() {
  char buf[101];
  char name[101];
  char *argv[101];
  int argc = 0;
  gets(name);
  gets(buf);
  while(buf[0] != 0) {
    argv[argc++] = strdup(buf);
    gets(buf);
  }
  argv[argc] = NULL;
  execv(name, argv);
}
```

Существует также вариант с суффиксом `"p"` ;

Вопросы

1. Как сказано выше, при системном вызове `exec` и его вариантах в виде семейства функций `execv*` часто требуется заказывать память у системы под аргументы. Например, программа `213.cc` заказывала память функцией `strdup`. Стоит ли в таком случае освобождать память под заказанные строки для аргументов и под массив указателей, если `exec*` — функция без возврата?

   Освобождать память стоит. Если функция `exec*` *завершится ошибкой, то вызывающий процесс оставит ресурсы занятыми. Общий полезный принцип — если ресурс не нужен, отдай его как можно раньше.*

4 Основы ввода/вывода. Взаимодействие процессов

4.1 Понятие дескриптора файла. Системные вызовы `write` и `read`

Все системные вызовы для исполнения ввода/вывода, независимо от места назначения — файл, экран, сеть — используют дескрипторы файлов. Дескриптор файла — небольшое неотрицательное число, связанное с каким-либо открытым файлом (вывод на терминал и ввод с клавиатуры в терминале, кстати, тоже управляются открытыми файлами).

При запуске любого процесса гарантируется, что он уже имеет три открытых файла. Они соответствуют файловым дескрипторам с номерами:

- `0` — стандартный ввод, обычно назначен на клавиатуру, но, как мы уже видели, может быть и перенаправлен.
- `1` — стандартный вывод, обычно назначен на терминал и тоже может быть перенаправлен.
- `2` — стандартный вывод ошибок, тоже обычно назначен на терминал. Он полезен в тех случаях, если стандартный вывод уже перенаправлен, а процессу требуется вывести на терминал диагностическое сообщение, например, сообщение об ошибке.

Системный контекст процесса содержит таблицу открытых файлов этого процесса. Можно считать в достаточном приближении, что дескриптор есть номер записи в этой таблице.

Простейшая программа, использующая системный вызов `write` может выглядеть так:

```
#include <unistd.h>
int main() {
  write(1, "Hi again\n", 9);
}
```

Сам системный вызов `write` имеет следующий прототип:

```
int write(int fd, const void *buf, int buflen);
```

Первый аргумент этого системного вызова — дескриптор файла, в который будет производиться запись. Напоминаем, что стандартный вывод всегда при запуске процесса связан с каким-либо файлом.

Второй аргумент — указатель на буфер, содержащий данные, которые будут записываться в этот файл. Модификатор `const` показывает, что несмотря на то, что мы передаём указатель в системный вызов, данные в буфере изменяться не будут. Таким образом, в файл передаётся содержимое куска памяти, такой ввод/вывод часто называют *сырым*, *raw* в отличие от *форматированного*.

Третий аргумент — количество передаваемых байт. Если, например, выделенный размер области памяти, буфера, равен 4096 байтам, то в третьем аргументе допустимо любое число от 0 до 4096. Использование памяти за пределами буфера чревато ошибками, иногда трудно обнаруживаемыми.

В нашем случае мы выводим 9 байт из участка памяти, содержащего 10 байтов -

`'H', 'i', ' ', 'a', 'g', 'a', 'i', 'n', '\n', '\0'.`

Последний символ нам на экране не нужен, мы его и не выводим.

Системный вызов `write` возвращает, сколько байтов он смог записать. Скорее всего, наш системный вызов возвратит число `9`.

Второй важный системный вызов — `read`. Его прототип такой:

```
int read(int fd, void *buf, int buflen);
```

Основное отличие — изменилось направление передачи информации. Теперь информация извне поступает в процесс, соответственно, модификатора `const` теперь нет — область памяти для поступающей информации должна быть доступна по записи. Опять же, размер выделенного для приёма информации блока памяти должен быть не меньше количества запрашиваемых байт. Возвращаемое системным вызовом значение показывает количество байт, помещённых в буфер («сырых» байт).

```
#include <unistd.h>
int main() {
  char buf[16];
  int bytes = read(0, buf, 16);
  write(1, buf, bytes);
}
```

Отрицательное количество возвращённых этими системными вызовами байт свидетельствует об неуспешном завершении системного вызова. Распространённая причина — неверный дескриптор файла. Другая распространённая причина — нарушение полномочий. Например, может оказаться, что нельзя писать в файл стандартного ввода.

Системные вызовы `read` и `write` настолько важны и универсальны, что мы изучили пока не все их свойства. Но пользоваться ими мы уже можем.

Системный вызов `close` разрывает связь между дескриптором файла и самим файлом. Если известно, что процесс больше не будет пользоваться файлом, файл стоит закрыть.

Парный к нему вызов `open` имеет достаточно необычный прототип и сам по себе является одним самых сложных системных вызовов.

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

int open(const char *name, int flags, int permissions);
```

Аргумент `flags` служит для управления режимами работы. А их действительно много, что и делает этот системный вызов таким сложным. Давайте перечислим самые популярные режимы:

- направление передачи данных. Нужно указать хотя бы один из элементов.

  `O_RDONLY` — открыть файл только для чтения
  - `O_WRONLY` — открыть файл только для записи
  - `O_RDWR` — открыть файл для чтения и записи

  Остальные элементы необязательны и добавляются к обязательному через операцию побитового или `'|'`.

- `O_CREAT` — создаёт файл с указанным именем, если его не существовало.
- `O_EXCL` — в сочетании с флагом `O_CREAT` при существовании создаваемого файла возникает ошибка.
- `O_NDELAY` - переводит дескриптор файла в *неблокирующий* режим. Об этом позже.
- `O_APPEND` - операции записи в файл после открытия будут производиться в конец файла.
- `O_TRUNC` - файл при открытии будет усечён до нуля байтов. Режимы доступа к нему во возможности сохраняются.

Аргумент `permissions` аналогичен такому в системном вызове `chmod`, используется только при наличии флага `O_CREAT` и назначает вновь создаваемому файлу соответствующие биты защиты.

Системный вызов возвращает дескриптор файла, если операция прошла успешно, или `-1` в противном случае (как обычно, причина ошибки помещается в переменную `errno`).

Дескриптор файла есть единственный путь доступа к подсистеме ввода/вывода операционной системы. Каждый дескриптор формирует объект ядра, ответственный за ввод/вывод. UNIX-like операционный системы спроектированы таким образом, что под общим понятием *файл* понимается всё, для чего возможны операции ввода/вывода - файл на диске, сетевое соединение, графопостроитель, магнитная лента, жёсткий диск как устройство. Об этом будет подробнее далее. Для термина дескриптор файла существует неформальный эквивалент - *канал ввода/вывода* или просто *канал*. С использованием этого термина некоторые понятия формируются короче и понятнее: канал на запись, канал на чтение, неблокирующий канал (при использовании `O_NDELAY`). Мы тоже будем употреблять этот термин в дальнейшем.

Примеры:

```
int fdin = open("somefile", O_RDONLY);
int fdout = open("anotherfile", O_WRONLY | O_CREAT, 0666);
if (fdout == -1) {
  printf("Error while creating file anotherfile, errorcode=%d\n", errno);
}
```

Задания:

1. Напишите программу, выводящую на экран информацию из буфера, причём количество выводимой информации больше размера буфера. Поэкспериментируйте с количеством байтов. Например, при буфере в 10 элементов попытайтесь вывести 11, 15, 20, 100, 1000, 10000. Объясните результаты, если можете.

2. Какое значение возвратит системный вызов `write(999,"Hi" , 2);`, если он будет единственной строкой в программе? А `write(0,"Hi" , 2);`?

   Первый возвратит -1. Второй может возвратить и 2 и -1, точный ответ зависит от реализации.

3. Что обозначает дескриптор файла? Чему он соответствует в контексте ядра процесса?

4. Напишите программу, которая копирует стандартный ввод в стандартный вывод. Для простоты положите, что если системный вызов `read` возвратил нулевое значение, то копирование надо прекратить.

   Простейший пример:

   ```
   #include <unistd.h>
   int main() {
       char c;
       while (read(0, &c, 1) > 0)
           write(1, &c, 1);
   }
   ```

   Предупреждение: медленная программа!

5. Напишите программу, которая копирует содержимое одного файла в другой. Имена файлов возьмите из командной строки. Не копируйте, если файл для записи уже существует.

   Ошибки ввода/вывода не обрабатываются!

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

   int main(int argc, char **argv) {
       char c;
       if (argc != 3) {
           printf("Usage: mycp from_file to_file\n"); exit(1);
       }
       int fdin = open(argv[1], O_RDONLY);
       if (fdin < 0) {
           perror(argv[1]); exit(1);
       }
       int fdout = open(argv[2], O_WRONLY | O_EXCL);
       if (fdout < 0) {
           printf("File %s exists, skipping\n", argv[2]); exit(1);
      }
      fdout = open(argv[2], O_WRONLY);
      if (fdout < 0) {
          close(fdin); perror(argv[2]); exit(1);
      }
      char buf[4096];
   ```

```
        int rd;
        while ( (rd = read(fdin, buf, sizeof buf)) > 0)  {
            write(fdout, &buf, rd);
        }
        close(fdout);
        close(fdin);
    }
```

4.2 Системный вызов pipe

. Процессы, созданные по системному вызову `fork` исполняются параллельно и это иногда само по себе может быть полезным, даже если они никак не взаимодействуют друг с другом. Взаимодействующие процессы - это бóльшая сила. Оболочка имеет несколько способов запускать взаимодействующие процессы из командной строки. Это бывает крайне полезно. Рассмотрим пример.

Имеется программа `wc`, которая умеет считать символы, слова и строки в файлах, указанных в командной строке. Вызванная без аргументов эта команда считывает информацию со стандартного ввода.

```
$wc
Testing program
wc.
All OK.<Ctrl-D>
       3       5      28
$
```

`<Ctrl-D>` есть комбинация символов, воспринимаемая по умолчанию за конец файла стандартного ввода (ну нужно же каким-нибудь образом дать понять, что все данные введены?).

Программа `cat` в свою очередь выводит на стандартный вывод всю информацию, содержащуюся в файлах командной строки.

Каким образом можно посчитать, используя эти команды, сколько строк имеют все файлы в директории `/usr/include`? Например, таким:

```
$cat /usr/include/* | wc -l
72467
$
```

Символ `'|'` в командной строке есть *метасимвол*. Оболочка, встретив этот метасимвол порождает не один процесс, а два. Этот метасимвол разделяет аргументы программ, поэтому запускаются программы со следующими аргументами:

- cat /usr/include/*
- wc -l

Эти процессы, исполняемые параллельно, создают то, что мы называем *конвейером*, когда стандартный вывод первого процесса становится стандартным вводом второго.

Этот механизм запуска параллельных взаимодействующих процессов оказал очень большое влияние на набор системных программ. Большинство системных программ умеет работать со стандартным вводом и стандартным выводом, если у них не указаны аргументы, если они обнаруживают, что стандартный вывод не принадлежит терминалу (об этом чуть позже) или если указан особый аргумент в командной строке. Оказалось, что часто удобнее иметь несколько простых программ, взаимодействующих через конвейер, чем одну сложную. Например, если кому-то придёт в голову фантастическая мысль распечатать на принтере все строки из файлов директории, содержащие слово **good**, попутно заменив все слова **good** на слова **bad**, причём на каждой странице должны быть 54 строки, то это делается одной командой:

```
grep good /usr/include/*.h | sed s/good/bad/ | pr -l54 | lpr
```

В данном случае конвейер состоит уже из четырёх процессов.

Ну а теперь о системном вызове, который и позволяет организовать такой конвейер. Это `pipe`.

```
  int fd[2];
  int code = pipe(fd);
```

Этот системный вызов возвращает в массиве `fd` два значения (обратим внимание, что в аргумент должен быть передан указатель на область памяти, содержащую место по крайней мере для двух `int`). Эти значения затем будут использоваться в системных вызовах чтения/записи - `fd[0]` может использоваться только для чтения, `read`, а `fd[1]` - только для записи, `write`.

Мы уже упоминали, что системный вызов `fork` клонирует процесс настолько, что полностью копируется пользовательский контекст процесса и на основе системного контекста процесса создаётся новая копия. Все файлы, открытые в процессе-родителе остаются открытыми в процессе-ребёнке (на самом деле это поведение можно изменить для некоторых открытых файлов, но это пока не важно). Поскольку `pipe` создаёт два открытых файла, эти файлы останутся открытыми после системного вызова `fork`. Воспользуемся этим для создания пары взаимодействующих процессов:

```
//h31.cc
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
int main() {
  int fd[2];
  pipe(fd);
  pid_t pid = fork();
  if (pid == -1) {
    perror("fork failed");
  } else if (pid == 0) {
    write(fd[1], "Hello from child", 17);
  } else {
    char buf[512];
    int bytes = read(fd[0], buf, sizeof buf);
    printf("Received %d bytes from child: '%s'\n", bytes, buf);
  }
}

$c++ h31.cc
$a.out
Received 17 bytes from child: 'Hello from child'
$
```

При использовании `pipe` возникает вопрос: каким образом информация передаётся из процесса в процесс? Имеется ли промежуточное хранилище для передаваемых данных? Ответ заключается в том, что системный вызов `pipe` создаёт новый объект ядра операционной системы, доступный только посредством системных вызовов `read` и `write`. Этот объект часто реализуется в виде кольцевого буфера. В этом случае он имеет несколько подобъектов: кусок памяти фиксированного размера, указатель на начало считываемых данных и указатель на начало записываемых данных (рисунок).

Операция записи приводит к внесению информации в буфер и перемещению указателя записи в новую позицию. Если указатель записи «*догоняет*» указатель чтения, то записывающий процесс блокируется до освобождения хотя бы части буфера. (рисунок2)

Операция чтения передаёт информацию в считывающий процесс только тогда, когда указатель чтения «*отстаёт*» от указателя записи. В противном случае считывающий процесс тоже блокируется до заполнения хотя бы части буфера.

Объект, созданный `pipe`, существует до тех пор, пока имеются открытые его дескрипторы файлов. Как мы уже говорили, признаком хорошего тона является закрытие тех файлов, которые мы (больше) не будем использовать. Поэтому процесс, выбравший направление передачи информации (чтение или запись), обычно закрывает неиспользуемую пару.

```
  } else if (pid == 0) {
    close(fd[0]);
    write(fd[1], "Hello from child", 17);
  } else {
   close(fd[1]);
    char buf[512];
    int bytes = read(fd[0], buf, sizeof buf);
    printf("Received %d bytes from child: '%s'\n", bytes, buf);
  }
```

Если какой-либо из процессов закрывает свою сторону до того, как другой процесс передал или получил информацию, объект получает статус «вдовца». Чтение из такого объекта вернёт 0 байтов, запись в него вызовет ошибку (`"Broken pipe"`).

Теперь попробуем реализовать упрощённый вариант конвейера оболочки. Мы хотим, чтобы запущенная нами чужая программа передала нам информацию. Например, нам хочется получить в свой процесс информацию, которую выведет программа с командной строкой `ls -l /`

```
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
int main() {
  close(0);
  close(1);
  int fd[2];
  pipe(fd);
  pid_t pid = fork();
  if (pid == -1) {
    perror("fork failed");
  } else if (pid == 0) {
    close(fd[0]);
    execl("/bin/ls", "ls", "-l", "/", NULL);
  } else {
    char buf[4096];
    int bytes;
    close(fd[1]);
    while ((bytes = read(fd[0], buf, sizeof buf)) > 0) {
       fprintf(stderr, "Received %d bytes from child: '%s'\n", bytes, buf);
    }
  }
}
```

В этой программе применяется способ, который кажется немного хакерским, но он годится для всех Unix-like систем. Строки `close(0)` и `close(1)` кажутся достаточно непонятными, ведь мы закрываем файлы стандартного ввода и стандартного вывода! Однако, после этого дескрипторы файлов с номерами `0` и `1` становятся свободными, система начнёт их раздавать процессам, которым потребовались открытые файлы и системный вызов `pipe` начинает использовать их. `fd[0]` становится равным `0`, а `fd[1]` становится равным единице. Поскольку при `exec` все открытые файлы, если это не изменено явно, остаются открытыми, стандартный вывод (файл с дескриптором, равным `1`) для `/bin/ls` замыкается на файл `fd[0]` родительского процесса. Поскольку стандартный ввод в родительском процессе тоже закрыт, информацию на терминал мы выводим, используя дескриптор файла `2`, который связан с `stderr`.

Системный вызов `pipe` можно использовать только для взаимодействия родственных процессов. Важно, чтобы сам системный вызов был исполнен в процессе, являющемся предком,

может быть и не прямым, взаимодействующих процессов. Другими словами, его не получится использовать, если взаимодействующие процессы должны принять решение о взаимодействии в произвольный момент времени — `pipe` может использоваться только в плановом режиме и планировать возможное взаимодействие должен предок. Конвейер — один из примеров такого, заранее запланированного оболочкой взаимодействия.

Проблема системного вызова `pipe` в том, что объект, который создаётся этим системным вызовом, не имеет имени и, следовательно, каких-либо способов доступа к нему. Как мы помним, каналы, образуемые средством связи `pipe` удовлетворяют дисциплине `FIFO` — First In First Out. Если такому средству связи присвоить имя, то его можно будет использовать для обмена информацией точно таким же образом, как мы использовали средство связи, созданное `pipe`. Имя такого объекта можно заимствовать в пространстве имён объектов файловой системы. Тип такого объекта будет `FIFO-файл`, а создать его можно командой `mkfifo`:

```
$mkfifo fifofile
$ls -l fifofile
prw-r--r--  student student  0 Sep 16 09:24 fifofile
$
```

Имя, которое мы создали командой `mkfifo` становится именем объекта ядра операционной системы, представляющего именованное средство связи. Однако, сам объект средства связи пока ещё не создаётся. Для использования его требуется по меньшей мере две операции: один процесс должен открыть `FIFO`-файл на чтение, другой — на запись. Только при успешном завершении обеих операций средство связи становится активными и его можно использовать.

Создадим новое окно терминала и перейдём в нём в директорию, в которой мы создали `FIFO`-файл `fifofile`.

```
Окно1$ cat fifofile                    Окно2$ cat < fifofile
qwerty                                 qwerty<Ctrl-D>
Окно1$                                 Окно2$
```

Набирая текст в окне 2 и завершая каждую строку клавишей `<Enter>`, мы видим набранный нами текст в другом окне. Два процесса `cat` взаимодействуют между собой.

Из собственных программ такое взаимодействие тоже просто проиллюстрировать.

```
//h33w.cc
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>

int main() {
    int fd = open("fifofile", O_WRONLY);
    if (fd < 0) {
       perror("open fifo for writing");
    } else {
       write(fd, "Using fifo file", 16);
    }
}

$c++ h33w.cc -o h33w
$
```

В другом окне:

```
//h33r.cc
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
```

```
int main() {
    int fd = open("fifofile", O_RDONLY);
    if (fd < 0) {
       perror("open fifo for reading");
    } else {
       char buf[128];
       int bytes = read(fd, buf, sizeof buf);
       printf("Read %d bytes from FIFO: '%s'\n", bytes, buf);
    }
}

$c++ h33r.cc -o h33r
$
```

Запустите эти программы в произвольном порядке — в одном окне `h33r`, в другом — `h33w`.

Обратите внимание на то, что после запуска одной программы её процесс переходит в состояние ожидания события появления другого. И только после запуска двух происходят какие-либо действия.

Вопросы:

- Классифицируйте средство связи, которое применяется в `pipe`.

  Симплексное, косвенное, поточное, надёжное

- Строка `"Hello from child"` в программе `h31.cc` вроде бы состоит из `16` байтов. Тем не менее, в аргументе `write` мы сказали, что надо передать `17` байтов. Для чего? Что будет, если мы передадим `16` байтов?

  `17` байт передаст в буфер нулевой байт. При чтении нулевой байт считывается и помещается в буфер чтения. `printf` по нулевому байту видит конец конец строки. Если писать `16` байтов, то `printf` после чтения будет выводить всё до ближайшего нулевого байта.

- Ёмкость средства связи, реализующего `pipe` ограничена. Придумайте эксперимент, который позволяет определить ёмкость этого средства.

  Писать в `pipe` по одному байту, распечатывая счётчик при успехе. Зависнет на очередной записи.

4.3 Особенности неблокирующего ввода/вывода

Если в системном вызове `open` во втором аргументе добавлен модификатор `O_NDELAY`, то ввод/вывод, который будет производиться через этот дескриптор становится *неблокирующим*, или, как говорят, мы создаём неблокирующий канал. При работе с дисковыми файлами (точнее сказать, с обычными файлами, *plain files*) мы не увидим разницы. Различия начинаются при использовании неблокирующих каналов для взаимодействия процессов. Блокирующий ввод/вывод старается соблюдать точное количество байтов в запросах. Если один процесс запросил из блокирующего канала `100` байтов, то процесс переводится в состояние ожидания до тех пор, пока другой процесс не обеспечит ему эти `100` байтов (или не завершится преждевременно, мы видели это при изучении системного вызова `pipe`, когда один процесс запрашивал `512` байтов, другой передавал ему `17` и завершался). Таким образом, блокирующий ввод может служить и служит удобным средством и взаимодействия процессов, и их синхронизации.

При неблокирующем вводе/выводе процесс, который запросил `100` байт может получить:

- `-1`, если другой процесс уже завершил работу;
- `0`, если другой процесс ничего не записал в канал после последней передачи;
- от `1` до `99` байтов, если другой процесс записал в канал в совокупности меньше `100` байтов;

- `100`, если другой процесс записал в канал `100` байтов и больше.

Проще всего рассматривать неблокирующий канал как ограниченное сверху хранилище байтов.

- Процесс-читатель не может прочитать из хранилища больше того количества, которое там хранится. Каждое успешное чтение уменьшает хранимое количество. Операция чтения завершается немедленно после получения информации, процесс никого не ждёт. Процессу передают либо запрошенное, либо доступное количество байтов, в том числе и ноль.
- Процесс-писатель не может записать в хранилище информации больше оставшегося до полной ёмкости хранилища. Хранилище никогда не может переполнится. Процессу возвращают то количество информации, которое система смогла поместить в хранилище (включая ноль байтов, если хранилище заполнено).
- Обоим процессам возвращают `-1`, если хранилище разрушено, например, системным вызовом `close` с любой из сторон.

4.4 Взаимодействие процессов посредством операций ввода/вывода

4.4.1 Передача базовых типов данных

Программы `h33` обмениваются между собой строками, точнее сказать, наборами символов. Не всегда такой обмен удобен. Представим, что один процесс должен передать другому целое 32-битной число. Конечно, можно преобразовать его в строку символов (например, с помощью `sprintf(buf, "%d", x)`, передать как строку и преобразовать из строки символов в целое (например, через `y = atoi(buf)`, но это довольно неудобно, да и просто долго.

Для удобства дальнейшего изложения, иллюстрирующего взаимодействие процессов, мы будем пользоваться ещё одним общим включаемым файлом `h34.h`

```
//h34.h
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
```

Тогда новый вариант двух процессов, обменивающихся целым числом, будет следующим:

```
//h34w.cc
#include "h34.h"

int main() {
    int fd = open("fifofile", O_WRONLY);
    if (fd < 0) {
       perror("open fifo for writing");
    } else {
       int x = 123;
       write(fd, &x, sizeof x);
    }
}

//h34r.cc
#include "h34.h"

int main() {
    int fd = open("fifofile", O_RDONLY);
    if (fd < 0) {
       perror("open fifo for reading");
    } else {
```

```
        int y;
        int bytes = read(fd, &y, sizeof y);
        printf("Read %d bytes from FIFO: '%d'\n", bytes, y);
    }
}
```

Мы вспомнили, что `read` и `write` передают просто набор байтов, расположенный по адресу. Если мы обеспечим идентичность объектов на приёме и на передаче, то произойдёт просто усложнённое транспортными средствами копирование переменной `x` первого процесса в переменную `y` второго процесса[3]

4.4.2 Передача сложных структур данных. Выравнивание.

Более сложные данные, например, составные, можно передать в виде структур. Как раз включаемый файл может описать все общие между процессами структуры данных.

```
//h35.h
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

struct common {
  int x;
  double y;
  char z;
};
```

Тогда новый вариант двух процессов, обменивающихся целым числом, будет следующим:

```
//h35w.cc
#include "h35.h"

int main() {
    int fd = open("fifofile", O_WRONLY);
    if (fd < 0) {
       perror("open fifo for writing");
    } else {
       struct common c;
       c.x = 123;
       c.y = 999.999;
       c.z = '#';
       write(fd, &c, sizeof c);
    }
}

//h35r.cc
#include "h35.h"

int main() {
    int fd = open("fifofile", O_RDONLY);
    if (fd < 0) {
       perror("open fifo for reading");
    } else {
```

[3]Мы пока пользуемся тем фактом, что и пишущий, и читающий процессы исполняются на одной вычислительной системе. В действительности `FIFO`-файлы можно использовать и для передачи информации между вычислительными системами. В этом случае передача информации подобным образом может оказаться некорректной, если будут использованы различные по архитектуре вычислительные системы. Подробнее об этом будет разговор позднее.

```
        struct common d;
        int bytes = read(fd, &d, sizeof d);
        printf("Read %d bytes from FIFO: x=%d y=%g z=%c\n", bytes,
           d.x, d.y, d.z);
    }
}

$./h35w (в другом окне)
$./h35r
Read 24 bytes from FIFO: x=123 y=999.999 z=#
$
```

Не настораживает тот факт, что `sizeof (struct common)` оказалось 24 байта вместо ожидаемого 13? Должно настораживать. Структуры для эффективности обращения к данным иногда выгоднее выравнивать по удобным для этих данных адресам, например, на 64-битных платформах обычно выгоднее размещать каждое из полей структуры по адресам, кратным восьми (что мы и наблюдаем здесь). На некоторых архитектурах такое размещение не просто удобно, оно вынуждено, процессор просто не сможет обратиться к невыровненному элементу структуры. Выровненное или невыровненное размещение структур управляется флагами компилятора и директивами (`#pragma`) в тексте программы. Скомпилировав на одной из архитектур программу `h35w` с изменённой командной строкой следующим образом, мы получаем совершенно неверный результат:

```
$g++ h35w.cc -fpack-struct=1 -o h35w
$./h35w
Read 13 bytes from FIFO: x=123 y=7.4805e-313 z=?
$
```

Задания:

1. Попробуйте откомпилировать программу чтения с изменённым флагом. Программу записи компилируйте обычным образом. Изменились ли результаты?

4.5 Файловый ввод/вывод. Системный вызов lseek

Системным вызовом `open` можно открывать, конечно, не только `FIFO`-файлы. Гораздо чаще мы используем его для работы с «обычными» (plain) файлами. Разница заключается в том, что `FIFO`-файлы по сути есть точка входа для использования поточных средств связи, а для таких средств связи не имеется понятия «возврат», то есть, нельзя повторно считать уже полученную информацию (конечно, её можно запомнить в считывающем процессе и использовать повторно, но это будет уже называться *буферизацией*). Таким образом, мы можем разделить все файловые дескрипторы на обслуживающие «поточные» средства связи и на обслуживающие «обычные» файлы. Существуют термины *streamed* или *sequentional* для обозначения поточных средств связи и *direct* для обозначения средств связи, основывающихся на «обычных» файлах. «Обычные» файлы часто называют «дисковыми», полагая, что они располагаются на устройстве «диск»[4]. Строго говоря, лучше их называть располагающимися на устройствах `DASD - Direct Access Storage Drive`, или `DASD`-файлами. Их — абсолютное большинство в любой вычислительной системе. Только для них с каждым дескриптором файла в процессе связывается некая *позиция* в файле, то есть то место в файле, начиная с которого будет производиться операция чтения или записи. Эту позицию можно получить или изменить системным вызовом семейства `lseek`. Успешные операции чтения иди записи продвигают позицию в файле на нужное количество байт. Давайте посмотрим примеры.

`//h35.cc`

[4]Этот термин с течением времени становится всё более абстрактным. Если это было безусловно верным в один из периодов развития вычислительной техники (кстати, магнитные барабаны появились ещё раньше), то сейчас под термином «диск» может скрываться и магнитный диск, и часть оперативной памяти (`RAM-drive`), и устройство флеш-памяти, `SSD - Solid State Drive`.

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

int main() {
  int fd = open("testfile", O_RDWR | O_CREAT, 0666);
  if (fd < 0) {
    perror("testfile");
    return 1;
  }
  for (int d = 0; d < 1000; d++) {
    int q = d * 10;
    write(fd, &q, sizeof q);
  }
  close(fd);
}
```

В этой программе нет пока ничего необычного. Она создаёт файл с именем `testfile` и пишет в него подряд 1000 чисел - 0, 10, 20, ..., 3990. После исполнения программы мы обнаружим в текущей директории файл `testfile` размером, как и ожидалось, 4000 байт (если мы не запустили нашу программу на вычислительной системе с необычной архитектурой).

```
//h37.cc
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

int main() {
  int fd = open("testfile", O_RDWR);
  if (fd < 0) {
    perror("testfile");
    return 1;
  }
  off_t pos = 500 * sizeof(int);
  lseek(fd, pos,SEEK_SET);
  int q;
  read(fd, &q, sizeof q);
  printf("q at position %lld is %d\n", (long long)pos, q);
  lseek(fd, -sizeof(int), SEEK_CUR);
  q = 999;
  write(fd, &q, sizeof q);
  lseek(fd, pos, SEEK_SET);
  read(fd, &q, sizeof q);
  printf("q at position %lld is now %d\n", (long long)pos, q);
  close(fd);
}

$./a.out
q at position 2000 is 5000
q at position 2000 is now 999
$./a.out
q at position 2000 is 999
q at position 2000 is now 999
$
```

Мы воспользовались системным вызовом `lseek` для изменения текущей позиции в файле.[5] Первый вызов `lseek` устанавливает позицию на байт с номером 2000 (мы всё ещё считаем,

[5] Системный вызов `lseek`использует тип `off_t` для второго аргумента. На некоторых архитектурах этот тип

что целое число занимает, как обычно, 4 байта) с начала файла (об этом говорит параметр SEEK_SET. Операция считывания целого числа системным вызовом read заполняет это число значением 5000, что согласуется с предыдущей программой (2000 / 4 * 10 = 5000). Текущей позицией в файле становится 2004. Следующая операция lseek перемещает позицию на sizeof(int) назад от текущей позиции (SEEK_CUR), то есть, на 2000. В этой позиции производится операция записи целого числа 999 (на место числа 5000, бывшего до него). Последняя группа операций показывает нам, что с позиции 2000 читается уже число 999.

Второй запуск программы показывает нам то, что мы ожидали. Информация в файле сохранилась между вызовами программ. Она, конечно, сохранится и при выключении компьютера.

Ещё одним возможным значением третьего аргумента является константа SEEK_END, при которой позиция устанавливается от конца файла. Как и в случае с SEEK_CUR смещение (второй аргумент) может быть и положительным и отрицательным. Результирующая позиция, конечно, не может быть отрицательной (перед началом файла ничего не существует), но она может быть установлена за концом файла! Что при это происходит?

```
//h38.cc
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <fcntl.h>

int main() {
  int fd = open("testfile", O_RDWR);
  if (fd < 0) {
    perror("testfile");
    return 1;
  }
  off_t pos = 50000000 * sizeof(int);
  lseek(fd, pos,SEEK_SET);
  int q = 87654321;
  int bytes = read(fd, &q, sizeof q);
  printf("q at position %lld is %d, read bytes=%d\n", (long long)pos, q, bytes);
  lseek(fd, pos, SEEK_SET);
  q = 999;
  bytes = write(fd, &q, sizeof q);
  printf("q at position %lld is %d, written bytes=%d\n", (long long)pos, q, bytes);
  lseek(fd, pos, SEEK_SET);
  q = 12345678;
  bytes = read(fd, &q, sizeof q);
  printf("q at position %lld is now %d, read bytes=%d\n", (long long)pos, q, bytes);
  close(fd);
}

$./a.out
q at position 200000000 is 87654321, read bytes=0
q at position 200000000 is 999, written bytes=4
q at position 200000000 is now 999, read bytes=4
$./a.out
q at position 200000000 is 999, read bytes=4
q at position 200000000 is 999, written bytes=4
q at position 200000000 is now 999, read bytes=4
$
```

имеет 32 бита, на некоторых — 64 бита. Поскольку мы хотим написать переносимую на различные архитектуры программу и не знаем, сколько бит в типе данных off_t, то при выводе на печать значений такого типа мы преобразуем его к заведомо не меньшему типу long long, для вывода которого в функции printf используется формат %lld

Мы видим, что система совершенно не запрещает перемещать позицию за предел файла. Однако, любая операция чтения за пределами файла вернёт нам нуль прочитанных байтов (что логично). Операция записи в файл при этом, однако, разрешена. При этом файл расширяется до необходимого размера.

```
$ls -l testfile
-rw-r--r--- 1 student student 200000004 Oct 13 12:27 testfile
$
```

Файл заполняется нулевыми байтами от старого конца файла до начала области записи. Некоторые файловые системы в этом случае не хранят эти нулевые байты внутри файла а просто помечают, что содержимое какой-то области — нулевое. Такое представление информации называется *разреженным* хранением.

Мы видим, что системный вызов `lseek` придал `DASD`-файлам новое интересное свойство. Каждый байт такого файла может быть доступен и записан или прочитан. В нашей программе `h37.cc` мы использовали файл как постоянное хранилище из 1000 целых чисел, каждое из которых можно прочитать за две операции (`lseek+read`) или записать за две операции (`lseek+write`). Такое хранилище может быть прочитано или записано любой программой, имеющий соответствующие права на файл. Использовать такие возможности можно многими способами:

1. трактовать файл как большой массив, правда, достаточно медленный;
2. обмениваться информацией между программами, возможно не исполняющимися одновременно;
3. хранить произвольные данные между запусками одной программы.

Вопросы и задания.

1. Почему было сказано, что системный вызов `lseek` на ряде архитектур ограничен диапазоном не $2^{32}-1$, а $2^{31}-1$?
2. Файл открывается в программе, затем вызывается системный вызов `fork`. Один из процессов изменяет позицию в файле. Напишите программу определяющую, изменяется ли позиция в файле в другом процессе.
3. Напишите набор функций, имитирующий работу с большим массивом с использованием файла. Должны быть функции заказа/освобождения памяти, получение `i`-го элемента, запись `i`-го элемента. Тип данных псевдомассива выберите сами.

4.6 Файловая система. Объекты файловой системы. Операции с файловой системой

Пока мы знаем несколько разновидностей объектов файловой системы. Это - обычные файлы, директории и `FIFO`-файлы.

4.7 Организация файловой системы

@@

4.7.1 Жёсткие связи в файловой системе. Системные вызовы `link` и `unlink`

@@

4.7.2 Символические связи в файловой системе. Системный вызов `synlink`

@@

4.7.3 Системные вызовы `stat` и `lstat`

@@

4.7.4 Функции `opendir`, `readdir` и `closedir`

@@

5 Потоки

Давайте немного определимся с терминологией. *Процессор* — устройство, исполняющее машинные команды программы (то, что обычно хранится в сегменте `.text`). Команды извлекаются из *оперативной памяти*.

Если вычислительная система содержит несколько процессоров, то мы называем такую конфигурацию *многопроцессорной* или *мультипроцессорной*. Наиболее простая система состоит из нескольких наборов (процессор ↔ память), связанных с собой линиями связи. Если каждый из процессоров может обращаться только к своей памяти, а общение процессоров между собой происходит специальными командами ввода/вывода (обычно относительно медленными), то такая архитектура называется массово-параллельной. На каждом из компьютеров исполняется своя копия операционной системы.

Если обращения процессора к «своей» памяти и к «чужой» производятся одними и теми же машинными командами, а время доступа между ними различается, то такая архитектура называется `NUMA`, Non Uniform Memory Access. Обычно исполняется одна копия операционной системы а на каждом из процессоров — свои модули.

Если память общая для всех процессоров, то такая архитектура называется `SMP`, *Symmetric Multi Processing*. Важно, чтобы время обращения каждого процессора к памяти было примерно одинаковым. На всех процессорах исполняется одна копия операционной системы. Для предотвращения конфликтов при обращении разных процессоров к одной ячейке памяти должен присутствовать *арбитр* и предусмотрены *протоколы* взаимодействия процессоров и памяти.

Если вернуться к первым многопроцессорным компьютерам, то каждый процессор представлял собой отдельное устройство (сначала шкаф, затем плату а потом устройство, монтируемое на плате, жаргонное название *кристалл*, *чип*). Более новые технологии позволили помещать несколько процессорных устройств на один кристалл, монтируемый на плате. Такие архитектуры называются *многоядерными*. С точки зрения операционной системы все `SMP` системы, многопроцессорные они или многоядерные, выглядят одинаково, они обычно не имеют «любимых» процессоров и каждый процесс может исполняться на любом из процессоров (или на нескольких, как мы скоро увидим).

Понятие *вычислительного потока* первые появилось в массовой операционной системе в 80-х годах, в операционной системе `OS/2`, совместно написанной фирмами Microsoft и IBM (технологически куда более совершенной, чем появившаяся позже Windows, но по политическим соображениям брошенной Microsoft). Абсолютно все вычислительные системы, на которых могла исполняться система `OS/2` имели ровно одно вычислительное ядро, но концепция потоков оказалась крайне удобной и продуктивной в программировании. Начиная с 2006 года, когда массовые вычислительные системы стали по большей мере многоядерными (на серверах многопроцессорность использовалась уже много лет), эта концепция показала и свои мускулы — на многоядерных компьютерных стало работать просто удобнее — реакция системы на пользовательские действия улучшилась, компьютер стал выполнять больше действий за единицу времени.

Вспомним, что при системном вызове `fork` появляется ещё один процесс, клон родительского. Два независимо работающих процесса, каждый из которых обрабатывает свои собственные входные данные и формирует свои собственные выходные данные на двухпроцессорной вычислительной системе могут эффективно удвоить общую производительность. Такие задачи, однако, редки. В реальных задачах процессы должны взаимодействовать между собой. Каждый из процессов имеет собственное, изолированное от других, адресное пространство. В каждом из адресных пространств параллельно (или псевдопараллельно) исполняются наборы команд процессора.

Потоки, *threads*, есть исполнение нескольких наборов команд процессора в пределах одного и того же адресного пространства. Все современные операционные системы позволяют создавать потоки. Синтаксис создания потоков может отличаться в разных операционных системах, однако суть создания потока едина для всех них.

5.1 Порождение потока. pthread_create

Имеется *главная функция потока*, иногда называемая *рабочей функцией потока*. Специальный системный вызов запускает эту функцию с какими-то аргументами, после чего в процессе исполняется на один поток больше. Проще всего считать, что при запуске процесса всегда порождается ровно один поток, главный поток.

Для использования потоками в Linux (и других UNIX-like системах) имеется библиотека pthreads.

```
#include <pthread.h>
```

Главная функция потока, которая будет вызываться, должна удовлетворять следующему прототипу:

```
void *(*start_routine)(void *);
```

Пробираясь через дебри синтаксиса языка C, мы видим, что она должна принимать ровно один аргумент, указатель на что угодно (void *) и возвращать указатель такого же вида.

Включаемый файл pthread.h> предоставляет нам новый тип данных pthread_t, который будет использоваться для управления потоками. Сама функция создания нового потока имеет следующий прототип:

```
int pthread_create(pthread_t *thread,
    const pthread_attr_t *attr, void *(*start_routine)(void *),
    void *arg);
```

В первом аргументе передаётся указатель на существующую переменную типа pthread_t, в которую после успешного исполнения функции создания потока будет возвращён идентификатор потока. Второй аргумент пока пропустим, временно будем использовать NULL на его месте. Третий аргумент — имя главной функции потока. Четвёртый — указатель на блок аргументов, которые передаются в поток при его запуске. Он может быть равен NULL, если мы ничего не хотим передавать.

Давайте посмотрим, как этим всем пользоваться. Вот программа, которая создаёт простейший поток.

```
//h41.cpp
#include <pthread.h>
#include <unistd.h>
#include <stdio.h>

void *func(void *arg) {
    printf("Hi from thread\n");
    return NULL;
}

int main() {
   printf("About to launch a new thread\n");
   pthread_t thr;
   pthread_create(&thr, NULL, func, NULL);
   printf("Thread created\n");
}

$c++ h41.cpp -lpthread
$./a.out
About to launch a new thread
Thread created
$
```

Если мы несколько раз запустим эту программу, возможно, мы всё же получим сообщение ("Hi from thread"). Возможно, этого сообщения мы не получим ни разу. Разве мы неправильно

вызвали функцию `pthread_create`? Правильно. Дало в том, что наш процесс уже исполнял главный поток — тот, который был создан при запуске процесса и функции `main` и он тоже исполнялся вместе с порождённым. Как только главный поток завершился по достижению закрывающей фигурной скобки в `main`, завершился и весь процесс, включая все порождённые потоки.

Давайте дадим порождённому потока шанс что-либо вывести, пока главный поток процесса не завершился. Для этого добавим следующую строку перед завершающей фигурной скобкой в **main**:

```
usleep(1000000);
```

Мы познакомились ещё с одним системным вызовом, `usleep`, который переводит текущий поток в состояние ожидания на указанное количество микросекунд. Мы задержали главный поток процесса на одну секунду.

```
About to launch a new thread
Thread created
Hi from thread
```

Задание:

1. Попробуем уменьшать значение аргумента, передаваемого в `usleep`. Делайте это до тех пор, пока вывод дополнительной строки не исчезнет.

2. Какое наименьшее значение аргумента `usleep`, при котором новый поток исполняется? Объясните.

   В разных реализациях по разному. При `usleep(1)` все операционные системы устойчиво выводят строчку из потока. При `usleep(0)` часть версий `linux` и `MacOS` перестаёт выводить. Всё зависит от версии планировщика.

5.2 Завершение потока. `pthread_join`

С точки зрения временны́х операция порождения потока напоминает вызов `fork`, и в том и в другом случае рисунок один (@@привести). Ожидание порождённого процесса, `wait`, тоже имеет свою пару, (pthread_join). Если цель нашей простой программы была в иллюстрировании работы нового потока, то лучшим способом является вызов функции `pthread_join`.

```
int main() {
   printf("About to launch a new thread\n");
   pthread_t thr;
   pthread_create(&thr, NULL, func, NULL);
   printf("Thread created\n");
   pthread_join(thr, NULL);
}
```

Теперь всё работает безупречно. Первый аргумент `pthread_join` - идентификатор ожидаемого потока. Отличный от `NULL` второй потребуется, если главная функция завершающегося потока решил что-то вернуть по `return`. Об этом немного позже.

5.3 Взаимодействие потоков

5.3.1 Общие переменные.

Пока написанная нами главная функция потока довольно бесполезна. Хотя она и исполняется параллельно с основной, особо ничего полезного она не производит, так как она с ним не взаимодействует.

Поскольку функция потока работает вместе с главной функцией в одном адресном пространстве, наиболее простой способ взаимодействия (но не самый лучший!) — через общие переменные.

```
//h42.cc
#include <pthread.h>
#include <unistd.h>
#include <stdio.h>

int common;

void *func(void *arg) {
  common = 10;
}

int main() {
   common = 5;
   printf("Before new thread common=%d\n", common);
   pthread_t thr;
   pthread_create(&thr, NULL, func, NULL);
   printf("After thread start common=%d\n", common);
   pthread_join(thr, NULL);
   printf("After thread end common=%d\n", common);
}

$c++ h42.cc -lpthread
$./a.out
Before new thread common=5
After thread start common=5
After thread end common=10
$
```

Если запустить получившийся исполнимый модуль несколько раз, то вторая строка может выглядеть по-другому:

```
After thread start common=10
```

Мы столкнулись с серьёзной проблемой, которая нас будет преследовать всё время, пока мы работаем с потоками: при внешне одинаковых условиях запуска программы она может вести себя по-разному. Проблема в том, что мы используем одну и ту переменную в двух различных потоках одновременно. Её значение в каждый из моментов времени будет зависеть от независимых от нас факторов. Такое поведение называется *состоянием гонок* или *race conditions*. Если вспомнить теоретическую часть и применить к данной программе критерий Бернстайна (напомнить про него), то он покажет, что «данная совокупность активностей недетерминирована». Бороться с таким поведением мы научимся немного позже.

5.3.2 Возвращаемое значение из главной функции потока

Если использование глобальных переменных может быть чревато нарушением детерминизма, то может быть, использовать другие способы общения потоков? Если целью деятельности порождённого потока есть получение каких-либо результатов, то можно воспользоваться возвращаемым значением. Поскольку это просто указатель, который не может в себе хранить значительное количество информации, то этот указатель должен показывать на какой-либо участок памяти, принадлежащий именно этому потоку. Как мы уже знаем, каждый поток имеет собственный стек. Но стек потока разрушается при завершении главной функции потока, поэтому возвращать указатель на область памяти внутри стека потока — плохая практика. Использование глобальной, общей памяти возможно, но не имеет особого смысла, поскольку она доступна вызывающему потоку и без возврата каких-либо значений из вызываемого потока. Поэтому наиболее естественный сценарий возникает, когда вызываемый поток заказывает память у системы из кучи, заполняет его необходимыми значениями и возвращает указатель на этот блок в главной функции потока.

```
//h43.cc
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

void *func(void *arg) {
    printf("Hi from thread\n");
    int *p = new int[2];
    p[0] = 7; p[1] = -7;
    return p;
}

int main() {
   printf("About to launch a new thread\n");
   pthread_t thr;
   pthread_create(&thr, NULL, func, NULL);
   printf("Thread created\n");
   int *p;
   pthread_join(thr, (void **)&p);
   printf("p[0]=%d p[1]=%d\n", p[0], p[1]);
   delete [] p;
}
```

В данном сценарии происходит следующее:

1. Порождается новый поток, ему выделяется стек.
2. В новом потоке у операционной системы заказывается участок памяти. Указатель `p` (и только он) адресует этот участок.
3. Заказанный участок памяти заполняется возвращаемыми значениями. Знанием о этом указателе обладает только порождённый поток.
4. Поток завершает исполнение, возвращая указатель `p`. Стек потока разрушается, указатель `p` сохраняется в структурах операционной системы, связанных с потоком (`TCB`, `Thread Control Block`). Эти структуры требуют немного ресурсов (в отличии от стека потока), поэтому их сохранение до завершения функции `pthread_join` не создаёт нагрузку на систему.
5. Функция `pthread_join` возвращает сохранённый указатель. `TCB` вместе с копией указателя, там хранящегося, освобождается.
6. Вызывающий поток пользуется возвращённым указателем. Указатель больше не нужен и память возвращается в систему.

Этот способ передачи имеет два недостатка:

- он передаёт информацию только в одну сторону;
- он нарушает негласный принцип «ресурсных скобок»: именно тот, кто заказывает ресурсы у системы должен их возвращать в систему. Это, конечно не обязательный принцип, более того, иногда его приходится нарушать, но его соблюдение приводит к более стабильным программам.

5.3.3 Аргументы главной функции потока

Наиболее распространённый способ обмена информацией между потоками — передача информации через аргументы.

```
//h44.cc
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

void *func(void *arg) {
    printf("Hi from thread, arg=%d\n", *(int *)arg);
    *(int *)arg = 7;
    return NULL;
}

int main() {
   pthread_t thr;
   int q = 10;
   pthread_create(&thr, NULL, func, &q);
   pthread_join(thr, NULL);
   printf("q=%d\n", q);
}
```

Откомпилировав и исполнив программу мы получим

```
Hi from thread, arg=10
q=7
```

Аргумент, который передаётся в поток, становится точкой взаимодействия между порождающим и порождённым потоками. Глобальные переменные тоже, как будто, могут служить точками взаимодействия, но их основная беда в том, что они могут использоваться в программе на только взаимодействующими потоками, но и любым программным кодом. Как известно, сложность составной системы в первую очередь определяется количеством точек взаимодействия между ними. При передаче данных через аргументы у нас появляется ровно одна точка взаимодействия и проконтролировать её гораздо проще, чем все глобальные переменные.

Приведённый пример показывает, что при наивном использовании аргументов функции потока синтаксис смотрится достаточно неуклюже. Изящнее следующий вариант, при котором используется копия переданного указателя, приведённая к нужному типу:

```
void *func(void *arg) {
    int *p = (int *)arg;
    printf("Hi from thread, arg=%d\n", *p);
    *p = 7;
    return NULL;
}
```

Для уменьшения зависимости потоков по данным следует разбить все передаваемые аргументы на три группы: только входные, только выходные и входные/выходные. Можно сказать, что передавая адрес участка памяти, порождающий создаёт контракт между собой и в порождённым потоком. Общая практика — использование структур (`struct`) и классов (`class`).

```
//h45.cc
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>

struct thr_args {
   int   in;
   double inout;
   char out[32];
```

```
};

void *func(void *arg) {
    struct thr_args * p = (struct thr_args *)arg;
    printf("Hi from thread, in=%d, inout=%g\n", p->in, p->inout);
    p->inout = 543.;
    strncpy(p->out, "Greetings from func", sizeof p->out);
    return NULL;
}

int main() {
   pthread_t thr;
   struct thr_args lt;
   lt.in = 999;
   lt.inout = 345.;
   pthread_create(&thr, NULL, func, &lt);
   pthread_join(thr, NULL);
   printf("Args after calling func: inout=%g out='%s'\n", lt.inout, lt.out);
}

$c++ h45.cc -lpthread
$./a.out
Hi from thread, in=999, inout=345
Args after calling func: inout=543 out='Greetings from func'
$
```

5.4 Синхронизация потоков

Мы уже использовали синхронизацию потоков, когда вызывали `pthread_join` в функции `main`. В этой точке происходила «встреча» всех порождённых потоков и функции `main` у *барьера*. *Барьером* мы называем точку, в которой встречающиеся потоки приостанавливают исполнение до тех пор, пока не появятся все из них. Мы заметили, что отсутствие барьера приводило к недетерминированному поведению. Вот другой пример.

```
//h46.cc
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

struct thr_args {
   int   in;
   int   inout;
};

void *func(void *arg) {
    struct thr_args * p = (struct thr_args *)arg;
    for (int i = 0; i < p->in; i++) {
        p->inout++;
    }
    return NULL;
}

int main(int argc, char **argv) {
   pthread_t thr;
   struct thr_args lt;
```

```
   int N = argc > 1 ? atoi(argv[1]) : 100000;
   lt.in = N;
   lt.inout = 0;
   pthread_create(&thr, NULL, func, &lt);
   for (int i = 0; i < lt.in; i++) {
      lt.inout++;
   }
   pthread_join(thr, NULL);
   printf("inout=%d, must be %d\n", lt.inout, N*2);
}

$c++ h46.cc -o h46 -lpthread
$./h46 10
inout=20, must be 20
$./h46 100
inout=200, must be 200
$./h46 1000
inout=2000, must be 2000
$./h46 10000
inout=20000, must be 20000
$./h46 100000
inout=119496, must be 200000
$
```

В этой программе мы впервые используем командную строку для своих целей. Аргументом программы является число повторений цикла. Запуск программы показывает, что небольшие значения счётчика приводят к ожидаемым результатам, а вот при преодолении некой границы результаты становятся неверными. Значение этой границы зависит от многих факторов, в том числе от быстродействия вычислительной системы, на которой исполняется программа, но факт, что программа ведёт себя неправильно. Одновременный доступ к переменной `inout`, доступной обоим потокам, приводит к недетерминированному поведению (критерий Бернстайна утверждает, что оно может быть).

Вопросы:

1. Почему при малых значениях числа повторений цикла состояние гонок не наблюдается (или наблюдается не всегда)?

   Это зависит от планировщика потоков. Порождение потоков занимает определённое время. Если главный поток успел к этому моменту времени завершить свой цикл, то конфликтов по записи больше не наблюдается.

5.4.1 `pthread_mutex`

Для предотвращения конфликтов можно определить операцию изменения переменной `inout` как операцию, входящую в критическую секцию. Может быть, можно использовать простой код, основанный на общих переменных? Например, такой:

```
//h46a.cc
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

struct thr_args {
   int   in;
   int   inout;
   int   *lock;
};
```

```
void *func(void *arg) {
    struct thr_args * p = (struct thr_args *)arg;
    for (int i = 0; i < p->in; i++) {
        while (*p->lock != 0)
           ;
        *p->lock = 1;
        p->inout++;
        *p->lock = 0;
    }
    return NULL;
}

int main(int argc, char **argv) {
   pthread_t thr;
   struct thr_args lt;
   int N = argc > 1 ? atoi(argv[1]) : 100000;
   lt.in = N;
   lt.inout = 0;
   lt.lock = 0;
   pthread_create(&thr, NULL, func, &lt);
   for (int i = 0; i < lt.in; i++) {
       while (lt.lock != 0)
           ;
       lt.lock = 1;
       lt.inout++;
       lt.lock = 0;
   }
   pthread_join(thr, NULL);
   printf("inout=%d, must be %d\n", lt.inout, N*2);
}
```

Откомпилировав и исполнив программу, мы получим примерно следующее:

```
$./h46 100000
inout=165345, must be 200000
$
```

Увы, от конфликтов мы так и не избавились.

Использование программных замков имеет два значительных ограничения:

1. пока один из потоков находится внутри критической секции, второй поток ожидает, используя процессорные ресурсы. Процессор обычно может пригодиться для более полезной работы. Чем дольше находится поток внутри критической секции, тем больше непроизводительный расход процессорного времени. Если вычислительная система одноядерная (однопроцессорная), то ожидание замка — совершенно бессмысленное и абсолютно непроизводительное расходование одного из ценнейших ресурсов, ведь на одноядерной архитектуре изменение замка другим потоком становится возможным только после переключения потоков, которое произойдёт после исчерпания кванта времени опрашивающим потоком.

2. если после завершения цикла и перед присвоением

   ```
   lock = 1;
   ```

   произойдёт переключение процессов, то вся конструкция разваливается. Для успешного функционирования переменной замка требуется, чтобы операции ожидания и присвоения были атомарными. Большинство современных процессоров имеет такие атомарные инструкции.

Так что не будем придумывать велосипед и вернёмся к использованию готовой структуры данных из библиотеки `pthreads`. Библиотека `pthread` имеет средство для создания критической секции и для операций с ней. Тип данных `pthread_mutex_t` служит как раз для этого.

Мы будем применять слово мьютекс. Это слово — транслитерация со слова *mutex - mutual exclusive*), к сожалению, в русском языке ничего подобного нет.

Вот пример использования мьютексов.

```
//h47.cc
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

struct thr_args {
   int   in;
   int   inout;
   pthread_mutex_t mutex;
};

void *func(void *arg) {
    struct thr_args * p = (struct thr_args *)arg;
    for (int i = 0; i < p->in; i++) {
        pthread_mutex_lock(&p->mutex);
        p->inout++;
        pthread_mutex_unlock(&p->mutex);
    }
    return NULL;
}

int main(int argc, char **argv) {
   pthread_t thr;
   struct thr_args lt;
   int N = argc > 1 ? atoi(argv[1]) : 100000;
   lt.in = N;
   lt.inout = 0;
   pthread_mutex_init(&lt.mutex, NULL);
   pthread_create(&thr, NULL, func, &lt);
   for (int i = 0; i < lt.in; i++) {
      pthread_mutex_lock(&lt.mutex);
      lt.inout++;
      pthread_mutex_unlock(&lt.mutex);
   }
   pthread_join(thr, NULL);
   pthread_mutex_destroy(&lt.mutex);
   printf("inout=%d, must be %d\n", lt.inout, N*2);
}
```

Недостаточно объявить переменную типа `pthread_mutex_t`, перед использованием надо её обязательно инициализировать (`pthread_mutex_init`).

Функкция `pthread_mutex_lock`, как говорят, *захватывает* мьютекс. Другой вызов этой функции во втором потоке заблокирует второй поток до тех пор, пока первый поток не *освободит* мьютекс. Таким образом ровно один поток имеет доступ к образовавшейся таким образом критической области. В нашем примере критическая область состоит из одной операции — увеличения общей переменной на единицу и она *ограждается* одним и тем же мьютексом (одним и тем же! это — обязательно!) от одновременного изменения в двух потоках.

Теперь наша программа работает правильно. Но ... Медленно, **очень** медленно!

Если первый вариант, без синхронизации, исполнялся столько:

```
$time ./h46 100000
inout=119549, must be 200000

real 0m0.005s
user 0m0.003s
sys 0m0.002s
$
```

то второй исполняется столько:

```
$ time ./h47 100000
inout=200000, must be 200000

real 0m0.404s
user 0m0.054s
sys 0m0.354s
$
```

Замедление программы оказалось ну оооочень большим. Операция, которую мы поместили в критическую секцию, сама по себе производится очень быстро, а вот операции синхронизации обычно требуют достаточно много ресурсов.

Примитивы синхронизации можно разделить на примитивы с активным ожиданием и с ожиданием события.

Первый тип примитивов синхронизации основывается на возможности потока совершать атомарные транзакции с памятью. Все многопроцессорные архитектуры содержат в своём наборе команд атомарные операции увеличения переменной или обмена переменных. Несколько потоков должны использовать одну и ту же переменную для синхронизации между собой. Если первый поток захватил примитив синхронизации, связанный с этой переменной, то второй поток будет исполнять операцию ожидания в цикле до тех пор, пока первый поток на освободит её. Из этого факта следует несколько выводов.

- *на одноядерной архитектуре такая конструкция бесполезна, так как при исполнения цикла в блокированном потоке блокирующий поток не получит управления до момента контекстного переключения потоков (по таймеру или по другому событию) и процессорное время, затраченное ожидающим потоком, будет затрачено впустую (не выполняется условие прогресса).*

- *на многоядерной архитектуре время, затраченное блокированным потоком на процесс ожидания, также относится к непродуктивному.*

- *переключения контекста при таком механизме не происходит. Это позволяет улучшить производительность системы в целом при условии, что блокирующий поток не захватывает примитив синхронизации на значительное время, сравнимое с временем контекстного переключения потоков.*

Второй тип примитивов синхронизации основывается на событиях. Событие предоставляется операционной системы как часть общесистемного механизма планирования. Объект, связанный с событием, является частью ядра операционной системы и только операционная система может предоставить связанные с ним операции, например, ожидание или установку. Соответственно, из понимания механизмов работы примитивов синхронизации, связанных с событиями можно сделать следующие выводы :

- *этот механизм одинаково работает на одноядерной и многоядерных архитектурах. Операция ожидания примитива приводит к информировании операционной системы об этом факте. В ядре операционной системы в очередь потоков, ожидающих данное событие, добавляется поток, вызвавший данный примитив, после чего происходит перепланирование потоков и управление передаётся какому-либо активному потоку.*

- *обязательно происходит переключение контекста, на это затрачивается значительное время, которое не может быть отнесено на счёт продуктивного. Частое переключение контекста приводит к деградации производительности вычислительной системы в целом.*

Таким образом, можно разделить сферу применения примитивов активного ожидания и примитивов ожидания события: Примитивы активного ожидания полезны при исполнении кода, требующего небольшого промежутка времени для блокировки, например, операции вставки в очередь, удаления из очереди или изменения переменных. Использование примитивов активного ожидания противопоказано на одноядерных вычислительных системах и во всех случаях, когда время ожидания заранее неизвестно. Примитивы ожидания события полезны в тех случаях, когда время ожидания заранее неизвестно и когда время захвата примитива заведомо велико, например, при обслуживании операций ввода-вывода.

Вернёмся к нашей программе. Такая потеря скорости вряд ли кого либо устроит. Можно ли для данной задачи улучшить результат?

Вариант 1. Переносим точки синхронизации (или, как их ещё называют *мьютексные скобки*):

```
//h48.cc
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

struct thr_args {
   int   in;
   int   inout;
   pthread_mutex_t mutex;
};

void *func(void *arg) {
    struct thr_args * p = (struct thr_args *)arg;
    pthread_mutex_lock(&p->mutex);
    for (int i = 0; i < p->in; i++) {
        p->inout++;
    }
    pthread_mutex_unlock(&p->mutex);
    return NULL;
}

int main(int argc, char **argv) {
   pthread_t thr;
   struct thr_args lt;
   int N = argc > 1 ? atoi(argv[1]) : 100000;
   lt.in = N;
   lt.inout = 0;
   pthread_mutex_init(&lt.mutex, NULL);
   pthread_create(&thr, NULL, func, &lt);
   pthread_mutex_lock(&lt.mutex);
   for (int i = 0; i < lt.in; i++) {
      lt.inout++;
   }
   pthread_mutex_unlock(&lt.mutex);
   pthread_join(thr, NULL);
   pthread_mutex_destroy(&lt.mutex);
   printf("inout=%d, must be %d\n", lt.inout, N*2);
}
```

```
$time ./h46 100000000
inout=105273894, must be 200000000

real 0m1.131s
user 0m2.239s
sys 0m0.003s

$time ./h48 100000000
inout=200000000, must be 200000000

real 0m0.375s
user 0m0.372s
sys 0m0.002s
$
```

Время исполнения резко уменьшилось. Однако, если внимательно посмотреть на исполняемый код, мы обнаружим, что эффективно перевели нашу программу из исполнения двух задач параллельно в исполнение этих же задач строго последовательно. Тем не менее, мы видим, на первый взгляд, непонятное событие: время последовательного исполнения двух фрагментов кода оказывается в данном случае в три раза *меньше*, чем при параллельном исполнении тех же фрагментов на многоядерной вычислительной системе.

Вопрос:

1. Чем можно объяснить тот факт, что в данном случае последовательный вариант программы исполняется быстрее параллельного? Считаем, что вычислительная система обладает достаточным количеством вычислительных ядер для параллельного исполнения всех потоков.

   Оба потока интенсивно используют одну и ту же область памяти. Это приводит к аппаратным конфликтам и процессор вынужден ожидать завершения операций записи в область памяти в одном потоке перед тем, как завершить операцию чтения в другом. Эти конфликты сильно замедляют исполнение машинных команд. Более подробно эта и другие проблемы обсуждаются в курсе Теория и практика многопоточного программирования, который автор читает на 3-м и 4-курсе ФУПМ. Материалы лекций и семинаров можно найти на сайте www.babichev.org

В данной задаче можно воспользоваться методом расслоения конфликтных переменных. Каждой переменной назначается своя область памяти (*декомпозиция*), после исполнения потоков проводится операция *редукции*, то есть, сборка (*композиция*) итогового значения из вычисленных локальных.

```
//h47.cc
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

struct thr_args {
   int   in;
   int   inout;
   pthread_mutex_t mutex;
};

void *func(void *arg) {
    struct thr_args * p = (struct thr_args *)arg;
    int q = 0;
    for (int i = 0; i < p->in; i++) {
        q++;
```

```
    }
    pthread_mutex_lock(&p->mutex);
    p->inout += q;
    pthread_mutex_unlock(&p->mutex);
    return NULL;
}

int main(int argc, char **argv) {
   pthread_t thr;
   struct thr_args lt;
   int N = argc > 1 ? atoi(argv[1]) : 100000;
   lt.in = N;
   lt.inout = 0;
   pthread_mutex_init(&lt.mutex, NULL);
   pthread_create(&thr, NULL, func, &lt);
   int q = 0;
   for (int i = 0; i < lt.in; i++) {
      q++;
   }
   pthread_mutex_lock(&lt.mutex);
   lt.inout += q;
   pthread_mutex_unlock(&lt.mutex);
   pthread_join(thr, NULL);
   pthread_mutex_destroy(&lt.mutex);
   printf("inout=%d, must be %d\n", lt.inout, N*2);
}

$time ./h49 100000000
inout=200000000, must be 200000000

real 0m0.240s
user 0m0.430s
sys 0m0.002s
$
```

Применение операции расслоения позволило добиться верного результата и хорошо распараллелить исполнение программы.

5.4.2 Тупики

Если мы используем только один мьютекс для защиты одного ресурса, то мы можем наблюдать некоторую деградацию производительности. Если два потока используют больше, чем один ресурс, то возможны и другие проблемы.

```
//h410.cc
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

struct thr_args {
   pthread_mutex_t m1;
   pthread_mutex_t m2;
};

void *func(void *arg) {
    struct thr_args * p = (struct thr_args *)arg;
```

```
    printf("func: about to lock mutex1\n");
    pthread_mutex_lock(&p->m1);
    printf("func: mutex1 locked, about to lock mutex2\n");
    pthread_mutex_lock(&p->m2);
    printf("func: mutex2 locked too, working\n");
    pthread_mutex_unlock(&p->m2);
    printf("func: mutex2 unlocked\n");
    pthread_mutex_unlock(&p->m1);
    printf("func: mutex1 unlocked\n");
    return NULL;
}

int main() {
   pthread_t thr;
   struct thr_args lt;
   pthread_mutex_init(&lt.m1, NULL);
   pthread_mutex_init(&lt.m2, NULL);
   pthread_create(&thr, NULL, func, &lt);
   printf("main: about to lock mutex2\n");
   pthread_mutex_lock(&lt.m2);
   printf("main: mutex2 locked, about to lock mutex1\n");
   pthread_mutex_lock(&lt.m1);
   printf("main: mutex1 locked too, working\n");
   pthread_mutex_unlock(&lt.m1);
   printf("main: mutex1 unlocked\n");
   pthread_mutex_unlock(&lt.m2);
   printf("main: mutex2 unlocked\n");
   pthread_join(thr, NULL);
   pthread_mutex_destroy(&lt.m1);
   pthread_mutex_destroy(&lt.m2);
   printf("all done\n");
}
```

Скомпилировав и несколько раз запустив программу `h410` мы неизбежно получим следующий результат:

```
$./h410
main: about to lock mutex2
main: mutex2 locked, about to lock mutex1
func: about to lock mutex1
main: mutex1 locked too, working
main: mutex1 unlocked
main: mutex2 unlocked
func: mutex1 locked, about to lock mutex2
func: mutex2 locked too, working
func: mutex2 unlocked
func: mutex1 unlocked
all done
$./h410
main: about to lock mutex2
func: about to lock mutex1
main: mutex2 locked, about to lock mutex1
func: mutex1 locked, about to lock mutex2
@@@@@ Нет ответа <Ctrl-C>
$
```

Программа попала в состояние тупика: главный поток ждёт мьютекс `m1`, который захвачен

порождённым потоком, а тот ждёт мьютекс m2, который захвачен главным. Никто уступать не собирается, поэтому — пат, тупик.

Приведённый пример может показаться искусственным. Вот классическая задача «обедающие мудрецы». Имеется пять мудрецов, которые хотят пообедать. Сегодня подают суши, которые положено есть двумя палочками (есть руками — да вы что, мудрецы не опустятся до такого). На столе лежит пять палочек. Если мудрец завладел двумя палочками, то он съедает свою порцию, кладёт палочки и покидает трапезную. Этими палочками может воспользоваться другой мудрец (нигде не сказано, моют ли они палочки перед едой — похоже, что нет). Надо написать программу, моделирующую деятельность мудрецов, каждый мудрец представляем в виде потока, каждую палочку — в виде мьютекса.

```
//h411.cc
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

struct wiseman_arg {
   int rank;
   pthread_mutex_t *sticks; // Указатель на общий массив
};

void *wiseman(void *arg) {
    struct wiseman_arg * p = (struct wiseman_arg *)arg;
    printf("wiseman %d: about to take the left stick\n", p->rank);
    pthread_mutex_lock(&p->sticks[p->rank]);
    printf("wiseman %d: about to take the right stick\n", p->rank);
    pthread_mutex_lock(&p->sticks[(p->rank+1)%5]);
    printf("wiseman %d: eating\n", p->rank);
    pthread_mutex_unlock(&p->sticks[(p->rank+1)%5]);
    pthread_mutex_unlock(&p->sticks[p->rank]);
    printf("wiseman %d: the sticks dropped\n", p->rank);
    return NULL;
}

int main(int argc, char **argv) {
   struct wiseman_arg w[5];
   pthread_t thr[5];
   pthread_mutex_t sticks[5];
   for (int i = 0; i < 5; i++) {
     pthread_mutex_init(&sticks[i], NULL);
   }
   for (int i = 0; i < 5; i++) {
      w[i].rank = i;
      w[i].sticks = sticks;
   }
   for (int i = 0; i < 5; i++) {
      pthread_create(&thr[i], NULL, wiseman, &w[i]);
   }
   for (int i = 0; i < 5; i++) {
      pthread_join(thr[i], NULL);
   }
   for (int i = 0; i < 5; i++) {
      pthread_mutex_destroy(&sticks[i]);
   }
   printf("all done\n");
}
```

```
$./h411
wiseman 0: about to take the left stick
wiseman 1: about to take the left stick
wiseman 0: about to take the right stick
wiseman 3: about to take the left stick
wiseman 4: about to take the left stick
wiseman 2: about to take the left stick
wiseman 1: about to take the right stick
wiseman 3: about to take the right stick
wiseman 4: about to take the right stick
wiseman 2: about to take the right stick
@@@ Опять зависли <Ctrl-C>
$
```

Поскольку все мудрецы используют один общий массив мьютексов, в контекст каждого из потоков приходится передавать указатель на этот массив[6]

Как бороться с тупиками? Есть несколько способов.

Первый способ носит название *супервизорного*. Все потоки, которые договариваются о совместных ресурсах, каждый из которых защищается критической секцией, заводят ещё один мьютекс, *супервизор*, который нужно захватывать при любой попытке войти в любую из защищаемых критических секций. Для мудрецов потребуется ровно один мьютекс на пять палочек.

```
//h412.cc
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

struct wiseman_arg {
   int rank;
   pthread_mutex_t *sticks; // Указатель на общий массив
   pthread_mutex_t *supervisor;
};

void *wiseman(void *arg) {
    struct wiseman_arg * p = (struct wiseman_arg *)arg;
    pthread_mutex_lock(p->supervisor);
    printf("wiseman %d: about to take the left stick\n", p->rank);
    pthread_mutex_lock(&p->sticks[p->rank]);
    printf("wiseman %d: about to take the right stick\n", p->rank);
    pthread_mutex_lock(&p->sticks[(p->rank+1)%5]);
    printf("wiseman %d: eating\n", p->rank);
    pthread_mutex_unlock(&p->sticks[(p->rank+1)%5]);
    pthread_mutex_unlock(&p->sticks[p->rank]);
    pthread_mutex_unlock(p->supervisor);
    printf("wiseman %d: the sticks dropped\n", p->rank);
    return NULL;
}

int main(int argc, char **argv) {
   struct wiseman_arg w[5];
   pthread_t thr[5];
   pthread_mutex_t sticks[5];
   pthread_mutex_t supervisor;
```

[6]почему? см. вопросы к разделу

```
   for (int i = 0; i < 5; i++) {
     pthread_mutex_init(&sticks[i], NULL);
   }
   pthread_mutex_init(&supervisor, NULL);
   for (int i = 0; i < 5; i++) {
      w[i].rank = i;
      w[i].sticks = sticks;
      w[i].supervisor = &supervisor;
   }
   for (int i = 0; i < 5; i++) {
      pthread_create(&thr[i], NULL, wiseman, &w[i]);
   }
   for (int i = 0; i < 5; i++) {
      pthread_join(thr[i], NULL);
   }
   for (int i = 0; i < 5; i++) {
      pthread_mutex_destroy(&sticks[i]);
   }
   pthread_mutex_destroy(&supervisor);
   printf("all done\n");
}
```

К сожалению, этот метод, хотя и избавляет программу от тупиков, превращает её в строго последовательную. Все мудрецы едят строго по порядку, а ведь это не то, чего мы обычно ждём от многопоточных программ.

```
$./h412
wiseman 0: about to take the left stick
wiseman 0: about to take the right stick
wiseman 0: eating
wiseman 0: the sticks dropped
wiseman 1: about to take the left stick
wiseman 1: about to take the right stick
wiseman 1: eating
wiseman 1: the sticks dropped
wiseman 3: about to take the left stick
wiseman 3: about to take the right stick
wiseman 3: eating
wiseman 3: the sticks dropped
wiseman 2: about to take the left stick
wiseman 2: about to take the right stick
wiseman 2: eating
wiseman 2: the sticks dropped
wiseman 4: about to take the left stick
wiseman 4: about to take the right stick
wiseman 4: eating
wiseman 4: the sticks dropped
all done
$
```

Этот метод, скорее, применим для быстрого исправления уже обнаруженных проблем синхронизации.

Этот метод, кстати, он применялся при выпуске операционной системы Windows 95. Её очень торопились сдать к намеченному сроку и при её проектировании было допущено удивительное количество как красивых изящных решений, так и откровенных недочётов. Одна из подсистем Windows 95 называлась Win16 и была предназначена для исполнения «старых» программ, написанных для Windows 3.0 и Windows 3.1. Эта подсистема использовала обращения (что-то вроде системных вызовов) к подсистеме Win32 (например, для операций ввода/вывода). Поскольку часть структур данных Win16 была спроектирована так, что повторное

исполнение одного и того же кода в разных потоках приводило к неверной работе (как говорят, код был *нереентерабельный*), то требовалось оградить использование такого кода критическими секциями. Как уже говорилось, системных вызовов в Windows было очень много уже в то время, а в каждом системном вызове требовалось войти в какую-либо критическую секцию, исполнить код и выйти из критической секции. Команда разработчиков просто не успевала написать необходимый код, а на отладку его и обнаружение потенциальных тупиков, видимо, потребовалось бы совсем астрономическое время. Поэтому было принять решение сделать всю подсистему Win16 критической секцией. Переменную, которая управляет этой критической секцией назвали Win16Lock. Как корабль назовёте, так он и поплывёт. Система действительно часто висла при плохом поведении процессов внутри критической секции. Через какое-то время её догадались переименовать в Win16Mutex. Очевидны последствия принятия такого решения. Как следствие (не единственное, конечно), Windows 95 и её потомки в виде Windows 98 и Windows Me так и не получили поддержку многоядерных архитектур (правда, их в то время и не было) в то время, как более правильно спроектированная Windows NT имела такую поддержку с самого начала.

Второй способ — *иерархия ресурсов*. Если каждому ресурсу назначить какой-либо приоритет (на самом деле важно лишь, чтобы каждому из ресурсов соответствовало какое-либо число и все числа были различны) и обязать потоки захватывать менее приоритетный ресурс только после захвата более приоритетного, то состояния тупика можно избежать. В примере с мудрецами приоритетом ресурса может быть, например, номер мьютекса в массиве и меньший номер пусть будет иметь больший приоритет. Тогда все потоки, кроме потока номер 4 (последнего из мудрецов) соблюдают этот принцип, а последний — нарушает его, захватывая сначала четвёртый, затем нулевой мьютекс. Если мы поменяем порядок захвата, проблема тупика исчезнет.

```
void *wiseman(void *arg) {
    struct wiseman_arg *p = (struct wiseman_arg *)arg;
    int first = p->rank < 4 ? p->rank : 0;
    int second = p->rank < 4 ? p->rank+1 : 4;
    printf("wiseman %d: about to take first stick\n", first);
    pthread_mutex_lock(&p->sticks[first]);
    printf("wiseman %d: about to take second stick\n", second);
    pthread_mutex_lock(&p->sticks[second]);
    printf("wiseman %d: eating\n", p->rank);
    pthread_mutex_unlock(&p->sticks[second]);
    pthread_mutex_unlock(&p->sticks[first]);
    printf("wiseman %d: the sticks dropped\n", p->rank);
    return NULL;
}
```

```
$./h413
wiseman 0: about to take first stick
wiseman 1: about to take first stick
wiseman 2: about to take first stick
wiseman 3: about to take first stick
wiseman 0: about to take first stick
wiseman 1: about to take second stick
wiseman 2: about to take second stick
wiseman 3: about to take second stick
wiseman 4: about to take second stick
wiseman 3: eating
wiseman 3: the sticks dropped
wiseman 2: eating
wiseman 2: the sticks dropped
wiseman 1: eating
wiseman 1: the sticks dropped
wiseman 0: eating
wiseman 0: the sticks dropped
wiseman 4: about to take second stick
```

```
wiseman 4: eating
wiseman 4: the sticks dropped
all done
$
```

Этот способ сочетает простоту и надёжность и в настоящее время очень широко применяется, например, в программировании ядра операционных систем (где тупики совершенно фатальны).

Последний способ заключается в моделировании исполнения программы (или только её операций синхронизации) с целью определения наличия тупиков или доказательства их отсутствия. Такое моделирование бывает динамическим и статическим.

При динамическом моделировании программа, оттранслированная специальным транслятором, исполняется под контролем управляющей программы, которая собирает информацию о потоках исполнения и может диагностировать проблемы синхронизации. Это обычно крайне долгий процесс, программа исполняется в десятки и сотни раз медленнее, чем обычно, к тому же, по причине недетерминированности, одиночное исполнение может не обнаружить тупик в программе, которая его содержит. Это напоминает поиск зелёной вороны — её обнаружение заканчивает поиск, а обнаружение очередной чёрной вороны всё еще не может убедить исследователя в отсутствии зелёных.

При статическом моделировании исследуется текст программы, а не процесс её исполнения. Для сложных языков, таких, как C++, иногда проще составить статическую модель поведения, указав в ней только точки синхронизации. Например, описание проблемы мудрецов может выглядеть так:

```
shared mutex sticks[5];
thread Wisemen[5] {
    stick[rank].wait;    stick[(rank+1)%5].wait;    @eat;
    stick[(rank+1)%5].release; stick[rank].release; @sleep;
}
```

Описание транслируется, например, в сеть Петри, анализ которой обнаруживает наличие тупика в модели.

Вопросы:

1. Можно ли в контекст потока передать сам массив мьютексов? Если да, то как надо изменить контекст потока?

2. Что произойдёт, если вместо указателя на массив мьютексов мы передадим копию самого массива?

6 Взаимодействие и синхронизация процессов

Как мы уже знаем, неродственные процессы могут взаимодействовать между собой, используя ввод/вывод, например, через `pipe` или через `FIFO`-файлы. Это, как мы помним, косвенный поточный способ связи. Альтернативным ему является способ связи, основанный на сообщениях.

Для использования сообщений требуется создать средство связи для обмена сообщениями — системную очередь сообщений.

6.1 Подсистема IPC и с чем её едят

Подсистема `IPC` — Inter Process Communications — появилась в Unix-like системах в коммерческой системе System V от AT&T и, хотя она реализована на всех Unix-like системах, её по традиции продолжают называть `System V IPC` или `SysV IPC`. Вся подсистема требует использования нового включаемого файла

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

Разные компоненты подсистемы (а она состоит из нескольких компонентов — сообщения, разделяемая память и семафоры) потребуют свои включаемые файлы.

Ключевым понятием этой системы является *идентификатор IPC*, средство управления, похожее на дескриптор файла. Все системные вызовы управления ресурсами IPC требуют этот идентификатор примерно как дескриптор файла требуется для операций с файлами.

Этот идентификатор имеет тип данных key_t и Получить этот идентификатор можно следующим образом:

```
key_t key = ftok("some_existing_file", 0);
```

В аргументы этому системному вызову надо подать имя какого-либо существующего файла. Он должен быть доступен данному процессу и всем процессам, которые хотят установить между собой взаимодействие. Этот файл не должен изменяться между вызовами ftok разных взаимодействующих процессов. Возникают вопросы: 1. Что это за файл? 2. Будут ли процессы использовать его содержимое? 3. Требуется ли помещать в него какую-либо информацию? Ответы могут показаться странными, но по ним становится ясно, почему выбран именно такой механизм создания ключа. 1. Любой доступный существующий файл. 2. Не будут. 3. Не требуется.

На самом деле создателям системного вызова потребовался простой механизм, каким образом два независимых процесса могут обменяться одним и тем же числом, неизвестным заранее. В качестве такого числа здесь применяется номер файла (точнее сказать, номер его I-узла, об этом чуть попозже) в файловой системе. Поскольку каждый файл в файловой системе имеет уникальный номер, его имя однозначно этот номер идентифицирует; этот номер, если файл не изменялся, будет одним и тем же для различных процессов, поэтому процессы таким косвенным образом обменяются этим числом.

Второй аргумент — небольшое целое число. Это число должно быть общим для взаимодействующих процессов. Идентификатор IPC часто называют **ключом** (кстати, название об этом намекает: ftok - File To Key). Так мы и будем называть его в дальнейшем.

Резюме: для разумной деятельности с использованием IPC для всех процессов требуется один и тот же ключ, которые добывается вызовом ftok.

6.2 Сообщения

Все системные вызовы этой группы начинаются с префикса msg а их прототипы содержатся во включаемом файле sys/msg.h>

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

6.2.1 Системные вызовы msgget, msgsnd и msgrcv

Очередь сообщений создаётся системным вызовом msgget, сообщения посылаются системным вызовом msgsnd, а принимаются системным вызовом msgrcv. Поскольку аргументы и семантика этих системных вызовов достаточно сложны, рассмотрим пример из двух взаимодействующих процессов.

Для удобства наш проект будет состоять из трёх файлов - одного общего включаемого с именем h600.h и двух с исходными текстами программ h600a.cc и h600b.cc.

```
//h600.h
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <unistd.h>
#include <stdio.h>

struct common {
  long mtype;
  int  body;
};
```

```
// h600a.cc
#include "h600.h"

int main() {
  key_t key = ftok("h600.h", 0);
  int mid = msgget(key, IPC_CREAT | 0600);
  if (mid < 0) {
    perror("msgget"); return 1;
  }
  struct common msg;
  msg.mtype = 123;
  msg.body  = 456;
  int code = msgsnd(mid, (struct msgbuf *)&msg, sizeof(int),   0);
  if (code < 0) {
    perror("msgsnd");
  }
}


// h600b.cc
#include "h600.h"

int main() {
  key_t key = ftok("h600.h", 0);
  int mid = msgget(key, IPC_CREAT | 0600);
  if (mid < 0) {
    perror("msgget"); return 1;
  }
  struct common msg;
  int code = msgrcv(mid, (struct msgbuf *)&msg, sizeof(int), 123,  0);
  if (code < 0) {
    perror("msgrcv");
  } else {
    printf("msgrcv received message with body=%d\n", msg.body);
  }
}
```

Первая программа, `h600a.cc` вначале получает так необходимый для всех `IPC` операций ключ. Ещё раз обратим внимание на то, что в первом аргументе мы передаём имя обязательно уже существующего файла, доступного процессу (точнее говоря, для осуществления полезных действий он должен быть доступен всем взаимодействующим процессам).

Системный вызов `msgget` создаёт новую очередь межпроцессных сообщений или подключается к существующей. Первый аргумент вызова — только что полученный нами ключ `IPC`. Семантика второго аргумента очень напоминает таковую в системном вызове `open` и управляется несколькими флагами. Один из них — `IPC_CREAT`, который указывает на то, что если очередь сообщений должна быть создана, если не была создана до этого, в этом случае требуется также указать режим доступа к создаваемой очереди, те же самые 9 битов маски доступа. Если мы хотим использовать систему очередей (или, как выяснится потом, любые `IPC` операции) для внутрипрограммного общения, вместо ключа, получаемого по `ftok` можно использовать константу `IPC_PRIVATE` — очевидно, что в этом случае очередь сообщений будет недоступна из других, неродственных, процессов. Как обычно, при неудачной попытке создания очереди сообщений или подключения к существующей, системный вызов `msgget` возвращает отрицательное значение. а при удачной — идентификатор очереди (аналог дескриптора файла в файловом вводе-выводе).

Сами сообщения передаются с помощью системных вызовов `msgsnd` и `msgrcv`. `msgsnd` в

первом аргументе принимает идентификатор очереди. Второй аргумент — адрес передаваемого участка памяти. В отличии от системного вызова `write`, в котором участок памяти может быть любым и на содержимое которого не накладывается никаких ограничений, системный вызов `msgsnd` требует обязательно область памяти определённого формата. Самое главное, чтобы первым элементом передаваемой информации был элемент с типом `long`, который должен содержать обязательно положительное число, называемого *типом сообщения* и которое в дальнейшем будет использоваться для получения сообщений. Наличие такого понятия позволит нам в дальнейшем не задумываться о ряде нюансов и отдать на откуп операционной системе следующие операции:

- получить сообщение строго определённого типа;
- получить сообщение с номером типа, большим данного;
- получить любое сообщение.

Ценность такого поведения в том, что при получении сообщения с определённым номером, другие сообщение остаются в очереди и могут быть получены в другое время (и, возможно, другим процессом).

Скомпилируем первый исходный файл `h600a.cc` и запустим его:

```
$c++ h600a.cc -o h600a
$./h600a
$
```

Наша программа создала очередь сообщений и поместила туда одно сообщение типа `123`. Программа завершила работу, а что произошло с очередью сообщений? И, вообще, как узнать, имеются ли в системе какие-либо очереди сообщений?

Для управления очередями сообщений существует программа `ipcs` — *IPC Status*. Она знает о всех трёх средствах межпроцессного общения — очередях, разделяемой памяти и семафорах. Если её вызвать без аргументов, то она выведет на стандартный вывод состояние всех трёх средств (точнее говоря, подсистем). Если нас интересуют только очереди сообщений, можно вызвать её с ключом `-q` (от слова *Queue* — очередь).

```
$ipcs -q
------ Message queues --------
key              msqid      owner        rights used bytes messages
0x00024da8 0          student        600          4           1
```

Мы видим, что в системе имеется одна очередь сообщений, которая принадлежит нам и биты защиты которой равны `0600` (то значение, которое мы просили в `msgget`). В очереди одно сообщение, всего в очереди `4` байта.

Вызовем программу `h600a` ещё несколько раз и посмотрим на очередь:

```
$./h600a
$./h600a
$./h600a
$ipcs -q
$ipcs -q
------ Message queues --------
key              msqid      owner        rights used bytes messages
0x00024da8 0          student        600        16           4
```

Мы видим, что очередь сообщений содержит теперь `4` элемента общим размером `16` байтов.

Что же, пора из этой очереди что-либо прочитать. Программа `h600b` пытается прочитать из очереди одно сообщение с типом `123`.

```
$./h600b
msgrcv received message with body=456
$ ipcs -q
```

```
------ Message queues --------
key              msqid      owner        rights used bytes messages
0x00024da8 0          student        600        12           3
```

Сообщение получено процессом, он распечатал правильное значение. В очереди стало на одно сообщение меньше.

Повторим операции:

```
$./h600b
msgrcv received message with body=456
$./h600b
msgrcv received message with body=456
$./h600b
msgrcv received message with body=456
$./h600b
... Не отвечает ...
```

Если в очереди имеются требуемые сообщения, они передаются принимаемому процессу и удаляются из очереди (нужно заметить, что наша очередь использует дисциплину `FIFO`). Если в очереди не имеется сообщений требуемого типа, то процесс, запросивший сообщение, блокируется, что мы и видим.

Блокировка принимающего процесса при приёме сообщения — один из очень важных способов синхронизации процессов. Передача сообщения от процесса процессу позволяет не только передать и получить необходимую информацию, а и быть уверенным в том, что передающий процесс уже подготовил нужную порцию и передал её для дальнейшего использования.

Пусть принимающий процесс желает проверить, не пришло ли сообщение нужного типа, но не хочет блокировать своё исполнение, ожидая этого сообщения. Последний, пятый аргумент в `msgrcv` в таком случае следует заменить на `IPC_NOWAIT`. Если сообщения с нужным типом в очереди не содержится, то системный вызов `msgrcv` возвратит `-1`, а код ошибки в `errno` будет установлен в `EAGAIN`. Не стоит злоупотреблять таким способом приёма сообщений, особенно плохо дожидаться таким образом прихода нужного сообщения — если программа действительно не может продолжать исполнение без требуемых в сообщении данных, значит нужно было исполнить блокирующий вариант системного вызова.

Четвёртый аргумент `msgrcv` позволяет получать из очереди не только сообщения определённого типа (имеющие конкретный номер типа), но и сообщения нескольких типов. Можно назвать этот аргумент фильтром сообщений. Если он равен какому-либо числу, большему нуля (вспомним, что типами сообщений могут быть только такие числа), то фильтр устанавливается на сообщения исключительно этого типа. Такой фильтр мы уже рассмотрели.

Если аргумент равен нулю, то фильтр отключается. `msgrcv` получит первое сообщение из очереди, независимо от его типа (или, как говорят, голову очереди).

В следующем примере первый процесс порождает `9` сообщений с разными типами, а второй процесс, используя фильтр равный нулю и режим чтения без блокировки, получает все их:

```
// h600c.cc
#include "h600.h"

int main() {
  key_t key = ftok("h600.h", 0);
  int mid = msgget(key, IPC_CREAT | 0600);
  if (mid < 0) {
    perror("msgget"); return 1;
  }
  struct common msg;
  for (int i = 1; i < 10; i++) {
    msg.mtype = i;
    msg.body  = i*i;
    int code = msgsnd(mid, (struct msgbuf *)&msg, sizeof(int),   0);
```

```
    if (code < 0) {
      perror("msgsnd");
    }
  }
}


// h600d.cc
#include "h600.h"

int main() {
  key_t key = ftok("h600.h", 0);
  int mid = msgget(key, IPC_CREAT | 0600);
  if (mid < 0) {
    perror("msgget"); return 1;
  }
  struct common msg;
  int code;
  while ((code = msgrcv(mid, (struct msgbuf *)&msg, sizeof(int), 0,  IPC_NOWAIT)) >= 0) {
    printf("msgrcv received message with type %ld and body=%d\n", msg.mtype, msg.body);
  }
}

$./h600c
$./h600d
msgrcv received message with type 1 and body=1
msgrcv received message with type 2 and body=4
msgrcv received message with type 3 and body=9
msgrcv received message with type 4 and body=16
msgrcv received message with type 5 and body=25
msgrcv received message with type 6 and body=36
msgrcv received message with type 7 and body=49
msgrcv received message with type 8 and body=64
msgrcv received message with type 9 and body=81
$
```

Последний вариант фильтра — использование отрицательного числа в четвёртом аргументе. Заменим строку

```
  while ((code = msgrcv(mid, (struct msgbuf *)&msg, sizeof(int), 0,  IPC_NOWAIT)) >= 0) {
```

на строку

```
  while ((code = msgrcv(mid, (struct msgbuf *)&msg, sizeof(int), -5,  IPC_NOWAIT)) >= 0) {
```

Сохраним файл под именем `h600e.cc`, скомпилируем его в исполнимый файл `h600e`
Ещё раз заполним очередь программой `h600c` и запустим `h600e`:

```
$./h600c
$./h600e
msgrcv received message with type 1 and body=1
msgrcv received message with type 2 and body=4
msgrcv received message with type 3 and body=9
msgrcv received message with type 4 and body=16
msgrcv received message with type 5 and body=25
```

Фильтру удовлетворили все сообщения с типами, значения которых оказались меньшими или равными абсолютной величине фильтра, то есть, 5.

Повторный вызов `h600e` приводит к отсутствию подходящих значений в очереди:

```
$./h600e
$
```

Вызов `h600d`, программы, получающей все сообщения независимо от их типа приводит к предсказуемому результату:

```
$./h600d
msgrcv received message with type 6 and body=36
msgrcv received message with type 7 and body=49
msgrcv received message with type 8 and body=64
msgrcv received message with type 9 and body=81
$
```

6.2.2 Использование сообщений разного типа

Рассмотренные примеры позволили понять сущность передачи сообщений и использовали простейшую структуру данных для передачи.

Давайте решим немного более сложную задачу: должны передаваться сообщения двух разных форматов. В сообщении первого формата полезная часть будет состоять из двух целых чисел, а в сообщении второго формата — из строки символов, длиной до 128 байт. Для того, чтобы их различить потребуется использовать различные типы сообщений, пусть они будут равны 1 и 2.

С общими структурами данных вроде бы всё понятно, для сообщений каждого из типов мы создаём свою структуру (которую, как и раньше, накладываем на системную структуру `msgbuf`, то есть, первым полем структуры обязательно должен быть `long mtype`).

```
//h601.h
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

struct type1 {
  long mtype;
  int  x,y;
};

struct type2 {
  long mtype;
  char str[128];
};
```

С передачей сообщений тоже всё ясно, сообщения первого типа передаются как объекты `type1`, второго — как объекты `type2`.

```
//h601a.cc
#include "h601.h"

int main() {
  key_t key = ftok("h600.h", 0);
  int mid = msgget(key, IPC_CREAT | 0600);
  if (mid < 0) {
    perror("msgget"); return 1;
  }
  struct type1 msg1;
  msg.mtype = 1;
  msg.x  = 22;
```

```
  msg.y  = 33;
  int code = msgsnd(mid, (struct msgbuf *)&msg1, sizeof msg1 - sizeof(long), 0);
  struct type2 msg2;
  msg.mtype = 2;
  strcpy(msg2.buf, "Hello from h601");
  code = msgsnd(mid, (struct msgbuf *)&msg2, sizeof msg2 - sizeof(long), 0);
}
```

Обратите внимания, для определения информационной части сообщения мы использовали выражение вида

```
sizeof msg1 - sizeof(long),
```

что удобно, но может быть не совсем корректно в случае необычного выравнивания (вспомним пункт 4.4.2).

Если программа для получения информации из очереди явным образом использует фильтры для сообщений каждого из типов, то проблем не наблюдается (т. е. если в `msgsnd` запрашивается сообщение первого типа, то известно, что в этом сообщении будет передано конкретное количество байт и под это сообщение можно подготовить конкретную структуру-приёмник).

Проблемы возникнут в том случае, если мы захотим принимать сообщения без применения фильтра, то есть, все приходящие.

6.2.3 Удаление очереди сообщений

Очередь сообщений стоит удалить, если ни одна из программ не собирается (или не в состоянии) ей воспользоваться. Проблема заключается в том, что сценариев использования очередей сообщений может быть много — от многопоточной программы, использующей очереди сообщений для обмена информацией между потоками, до многопроцессных приложений с изменяющимся количеством активных процессов, обменивающихся сообщениями многих типов. Когда удалять очередь? Как только выяснилось, что она не понадобится самому последнему из использующих её процессов? А как установить этот факт? А что, если процессы при удалении общей очереди понадеялись друг на друга и она осталась неудалённой?

Очередь сообщений — объект ядра, который уничтожается при завершении работы вычислительной системы, то есть, очередь существует до перезагрузки. Это первый способ удаления очередей (немного напоминающей отсечение больной головы).

Второй способ — использовать уже знакомую программу `ipcs` с ключом `-q` для обнаружения неудалённой очереди. Из информации, выдаваемой этой программой требуется определить идентификатор очереди (`msgid`), после чего эту очередь можно удалить другой программой, `ipcrm`.

```
$ ipcs -q
------ Message queues --------
key        msqid      owner        rights used bytes messages
0x00024da8 0          student        600        12           3
$ipcrm msg 0
$
```

Третий способ, наиболее правильный, заключается использовании системного вызова `msgctl`.
@@

6.3 Разделяемая память. Системные вызовы `shmget`, `shmat`, `shmdt`

@@

6.3.1 Примеры использования разделяемой памяти

6.3.2 Ввод/вывод встречается с разделяемой памятью: файлы, отображаемые на память. Системные вызовы `mmap` и `munmap`

.

6.4 Семафоры. Системные вызовы `semget`, `semop`

Использование разделяемой памяти между процессами без синхронизации неизбежно приводит к уже знакомым проблемам условий гонок (*race conditions*). Примитивы синхронизации, которые мы изучили в разделе `Потоки`, здесь не подходят, так как они ориентированы на общее адресное пространство, а у нас каждый процесс имеет своё адресное пространство (за исключением разделяемой памяти, конечно). Технически можно, конечно, попытаться разместить `pthread_mutex` именно в этой разделяемой памяти, но это, во-первых неудобно, а во-вторых, именно для таких задач удобнее (и эффективнее) использовать семафоры.

6.4.1 Как семафоры Дейкстры соотносятся с IPC семафорами

6.4.2 Удаление семафоров

7 Сигналы

@@

8 Сети

@@