Exam

Раздел 1

1.

Стадии сборки

```
1. Препроцессинг - раскрытие директив препроцессора
2. Компиляция - преобразование кода в ассемблерный
3. Ассемлирование - преобразование кода в машинный
4. Линковка - связывание нескольких объектных файлов
```

Как увидеть, из каких этапов состоит сборка программы через g++?

```
g++ -v test.cpp # подробная информация, в т.ч. посмотреть этапы компиляции
/usr/lib/gcc/x86_64-linux-gnu/11/cc1plus # препроцессинг и компиляция
as # ассемблирование
/usr/lib/gcc/x86_64-linux-gnu/11/collect2 # линковка
```

Как получить из cpp-файла ассемблерный код? Как получить объектный файл?

```
g++ -E test.cpp # (> test_preprossessed.cpp)
g++ -S test.cpp # (> test.s ассемблер)
g++ -c test.s # (> test.o объектный файл)
g++ test.o
```

Чем объектный файл отличается от исполняемого (бинаря)?

Объектный файл отличается от бинарного (исполняемого файла) содержанием не разрешённых ссылок на функции. То есть его нельзя запустить, так как определение некоторых функций ещё не найдено.

Как дизассемблировать исполняемый файл?

```
objdump -d a.out - дизассемлирование, по бинарю восстановить ассемблер
```

2.

Что такое линковка?

До стадии линковки в файле могут быть объявления функций и переменных без определений. На стадии линковки связываются несколько объектных файлов, чтобы подставить определения в объявления.

```
readelf -s test.o # содержит UND
readelf -s a.out # пытается вместо UND подставить что-то конкретное
или
```

```
objdump -d test.o # можно увидеть заглушки, которые стоят на местах вызова фукнций
```

Что такое библиотеки (либы)?

Библиотека - скомпилированный файл формата ELF, содержащий символы с определениями

Покажите на примере простейшей программы, как вручную с помощью ld слинковать объектный файл со стандартной либой C++.

```
ld test.o /usr/lib/x86_64-linux-gnu/libstdc++.so.6 /usr/lib/x86_64-linux-gnu/libc.so.6 /usr/lib/gcc/x86_64-linux-gnu/11/crtend.o /usr/lib/gcc/x86_64-linux-gnu/11/crtbegin.o

./a.out - No such file or directory. Проблема с динамическим линковщиком
```

Как добиться, чтобы полученный исполняемый файл удалось запустить?

```
# нужно добавить путь к стандратному динамическому линковщику
ld test.o /usr/lib/x86_64-linux-gnu/libstdc++.so.6 /usr/lib/x86_64-linux-gnu/libc.so.6 /usr/lib/gcc/x86_64-linux-gnu/11/crtend.o /usr/lib/gcc/x86_64-linux-gnu/11/crtbegin.o -I /usr/lib64/ld-linux-x86-64.so.2
# ld: warning: cannot find entry symbol _start; defaulting to 00000000004010b0
./a.out
# segfault
```

- Линковщик поставил неправильно **точку входа** в программу, так как она не указана. В данном случае программа начинается с функции main, но до main должны создаться статические и глобальные объекты
- В случае отсутствия таких объектов тоже будет segfault, так как в конце main есть return, который некуда делать. Можно решить при помощи exit(0) до return.

3.

Что такое статическая и динамическая линковка, статические и динамические либы?

Статическая линковка - весь необходимый код из библиотек объединяется исполняемым кодом программы в один объектный файл, т.е. копируется.

```
g++ -static test.cpp
```

Можно увидеть в .symtab, что адреса проставляются на последней стадии сборки программы
Динамическая линковка - Код библиотеки не включается в исполняемый файл, а используется во время выполнения программы
Можно увидеть в .dynsym символы, которые будут подргружены в рантайме

Статические библиотеки - компоновка выполняется во время компиляции, встраивая исполняемый код в двоичный файл

Динамические библиотеки - компоновка выполняется динамически при запуске программы

Как принудительно сделать статическую линковку вместо динамической и зачем это может быть нужно?

```
g++ -static test.cpp
```

Может быть полезно для запуска программы на других устройствах, где могут не быть установлены нужные библиотеки.

Как посмотреть, от каких динамических либ зависит данный исполняемый файл?

```
ldd a.out
```

Как собрать программу с либой C++, расположенной по нестандартному адресу?

```
ldd a.out (увидим путь к libstdc++.so.6, если содержит например include
iostream)
cd /lib/x86_64-linux-gnu/
sudo mkdir test
sudo mv libstdc++.so.6 test/
cd test/
ls -l # увидим файл, на который ведет ссылка (настоящая либа)
cd ..
sudo mv libstdc++.so.6.0.30 test/
LD_LIBRARY_PATH=/lib/x86_64-linux-gnu/test ./a.out
```

Как создать свою динамическую либу и собрать программу с ее использованием?

```
1. g++ -shared lib.cpp -o libmyfunc.so (создание shared object)
2. подключаем header с объявлением функции
Вариант 1:
        3. sudo mv libmyfunc.so /usr/lib/
        4. g++ test.cpp -lmyfunc (порядок имеет значение)
Вариант 2:
        3. g++ test.cpp -lmyfunc -L.  (слинковать файл с либой libmyfunc.so,
которую нужно искать в текущей директории статическому линковщику (-L))
        4. LD_LIBRARY_PATH=. ./a.out
Вариант 3:
        3. g++ test.cpp -lmyfunc -L. -Wl,-rpath,/caos (динамический
линковщик будет         искать либу по указанному абсолютному пути)
```

Для чего нужна переменная LD_LIBRARY_PATH?

Это предопределенная переменная среды в Linux / Unix, которая задает путь, к которому компоновщик должен обращаться при связывании динамических библиотек / разделяемых библиотек. Можно задавать через список путей, отделенных двоеточием, где библиотеки следует искать перед тем, как их будут искать по стандартным путям. Полезно использовать в случае, если библиотеки расположены не по стандартному пути

Что такое rpath и как его использовать?

> rpath - параметр, который позволяет в бинарный файл зашить информацию о том, где нужно искать динамическую библиотеку

```
-Wl,-rpath,/home/liza/caos  # путь - абсолютный
#-Wl - передать на стадию линковки, через запятую перечисляется, что передать
```

Как посмотреть, какие вызовы библиотечных функций делает данная программа в ходе выполнения?

```
ltrace -p 27766 # процесс обязательно родительский
```

4.

Формат ELF

> Elf - Executable Linkable Format (файл, подлежащий либо выполнению, либо линковке). Файл ELF, как правило, является выходным файлом компилятора или линкера и имеет двоичный формат.

Какие 3 вида ELF-файлов существуют?

1. REL (Relocatable file) - неслинкованный файл
2. DYN (Shared object file) - динамическая либа
3. EXEC (Executable file) - файл со статической линковкой
4. DYN (Position-Independent Executable file) - readelf -a a.out
5. CORE (Core file)

Покажите по одному примеру каждого вида

```
1.
g++ -c test.cpp
readelf -h test.o # REL
2.
g++ -shared test.cpp -o lib.so
readelf -h lib.so # DYN
3.
g++ -static test.cpp
readelf -h a.out # EXEC
4.
readelf -h a.out # DYN (Position-Independent Executable file)
```

Из каких основных секций состоят ELF-файлы?

1. Заголовок (magic в хедере - байты, с которых начинается файл. Одинаковые для всех файлво формата ELF)
2. .text - бинарный код исполняемой программы
3. .data - проинициализированные статические переменные
4. .rodata - read only data
5. .bss - непроинициализированные статические переменные
6. .symtab - таблица символов

Какие есть утилиты для чтения содержимого ELF-файлов?

1. hexdump - чтение по байтам
2. readelf
3. objdump (-t таблица символов)
4. nm - выводит только таблицу символов

Для чего нужна утилита objcopy? Приведите пример использования.

> Утилита `objcopy` используется для манипуляции объектными файлами в форматах, поддерживаемых GNU Binutils (например, ELF, COFF и других). Она позволяет копировать содержимое объектных файлов, изменять их формат и производить модификации секций.

```
objcopy --strip-debug input.elf output.elf # убрать дебажную информацию
objcopy --dump-section .text=text.bin input.elf # извлечение данных из
секции .text
objcopy -O binary input.elf output.bin # конвертация файла в бинарный
```

5.

Что такое символы в терминах линковщика?

> **Символы** - сущности, которыми оперирует линковщик

Что такое манглирование и как (в общих чертах) оно работает?

Манглированные имена можно увидеть в таблице символов.

> Манглирование - процесс преобразования имен по специальным правилам между стадиями компиляции и линковки для дальнейшей работы линковщика

Пример работы:
_ZSt3cin

- _Z - символ, определенный пользователем
- St - std::
- 3 - количество символов в имени
- cin - имя

Как по манглированному имени восстановить исходное?

```
c++filt "имя"
nm -C a.out # выводит все символы с деманглированным именем
```

Как посмотреть список символов в данном ELF-файле?

```
readelf -s a.out
```

Что такое weak и strong символы?

Столбец **Bind** принимает значения LOCAL, GLOBAL, WEAK.
GLOBAL - объявления глобальных функций и переменных, видны из-под других translation unit'ов. (strong)

LOCAL - глобальные static функции и переменные, локальные
WEAK - GLOBAL, но если встретится такой же символ без пометки WEAK, то он перезатрется

Какая бывает видимость у символов, как сделать символы приватными для внешних файлов?

У символов бывает внешнее и внутреннее связывание. Символ с внешней связью виден всем единицам трансляции. С внутренней - нет.
static, примененное к глобальной функции/переменной, означает сделать ее видимой только из под данного translation unit'a.

Что такое релокации?

> **Релокации** — это процесс изменения адресов и указателей, которые присутствуют в исполнимом файле программы, для того чтобы программа могла быть правильно размещена в памяти при запуске.

В процессе **линковки**, линковщик должен решить, где в памяти будет размещена каждая переменная или функция. Затем линковщик должен обновить все заглушки в объектных файлах, чтобы они указывали на правильные адреса в итоговом исполнимом файле. Этот процесс и называется **релокацией**.

Пример:
Предположим, у нас есть два объектных файла — `main.o` и `foo.o`. В `main.o` есть вызов функции `foo()`, но линковщик еще не знает точный адрес этой функции, так как она находится в другом объектном файле.

1. Компилятор создает объектный файл `foo.o`, в котором функция `foo()` не имеет точного адреса, а просто метку.
2. Линковщик соединяет оба объектных файла в один исполнимый файл и решает, где в памяти будет находиться код функции `foo()`.
3. Линковщик заменяет метку на реальный адрес, например, `0x1000`, если `foo()` будет находиться по этому адресу в памяти.
4. Теперь, когда программа будет загружена в память, все вызовы функции `foo()` будут использовать этот новый адрес.

6.

Запуск программы. Что происходит при запуске бинаря до начала функции main, а также после ее окончания

Программа начинается со _start, где создаются и инициализируются статические и глобальные объекты. После main уничтожаются глобальные и статические объекты. В самом конце вызывается функция exit(int), которая корректно завершает процесс.

Как попросить g++ не генерировать функцию _start, а начинать программу сразу с main?

```
g++ -e main hello.cpp # функция _start сгенерируется, но выполнение начнется с main
```

Что произойдет, если запустить такую программу, и почему?

Программа упадет с сегфолтом, так как в main написан return, который делать некуда. Если же есть exit(0), но есть переменные, которые должны быть созданы и проинициализированы до main'a, то тоже сегфолт.

Как исправить код функции main, чтобы программа после нее корректно завершалась, если функция _start отсутствует?

При отсутствии переменных, которые должны быть созданы и проинициализированы до входа в main, нужно написать exit(0) в конце main

7.

Дебаг. Что такое сборка в дебаг-режиме и в чем ее отличие от обычной?

Дебаг позволяет отследить выполнение программы
Стандартно скомпилированный файл не содержит информацию о том, какие строчки где были. Дебажная сборка добавляет отладочные символы

```
gdb -g test.cpp # на выходе DWARF файл
objdump -d a.out # вывод дебажной информации
```

Как с помощью gdb отлаживать программу: поставить breakpoint на строчку кода или на функцию, делать шаги по строчкам (с заходом в функции и без него), выводить текущие значения переменных?

```
gdb -g test.cpp
gdb ./a.out
b [test.cpp:]6 # поставить breakpoint
b [test.cpp:]f # поставить breakpoint на функцию
run
n # next - шаг без захода в функцию
s # step - шаг с заходом в функцию
p count # print - вывести текущее значение переменной
```

Как посмотреть backtrace от текущего места исполнения?

```
bt
```

Что означает фраза “core dumped” и как с помощью gdb посмотреть содержимое coredump-файла после того, как программа упала?

> core dumped означает, что сделан dump ядра, то есть прежде чем убить программу, операционная система сохраняет состояние в специальный core файл

По умолчанию на ubuntu хранятся в /var/lib/apport/coredump, но не сохраняются (лимит core файла - 0)

- ulimit -c - позволяет узнать максимальный разрешенный размер core файлов
- ulimit -c unlimited - меняет максимально разрешенный размер
  Можно:

1. читать core файл при помощи readelf
2. если собрать cpp с флагом -g, то

```
gdb ./a.out path-to-core-file
# Program received signal SIGSEGV, Segmentation fault.
# 0x0000555555555327 in main () at test.cpp:6
# 6       std::cout << v[100000000];
```

8.

Что такое системный вызов (сисколл)?

> Системный вызов - функция, которая осуществляет взаимодействие между программой и операционной системой. Предоставляется операционной системой

Как посмотреть в режиме реального времени, какие сисколлы происходят во время работы какой-нибудь программы?

```
strace -p "proccess-id"
```

Как посмотреть мануал для данного сисколла?

```
man 2 read
man 2 write
```

Покажите использование сисколлов на примере read и write. Как использовать возвращаемые значения этих сисколлов? Как проверить, успешно ли был выполнен сисколл, а в случае неудачи узнать, какая ошибка произошла?

```
#include <unistd.h>
#include <cstdlib>
#include <string>

int main() {
  char buffer[100];
  auto taken = read(0, buffer, 100);
  if (taken == -1) {
    auto errno_str = std::to_string(errno);
    write(2, errno_str.data(), errno_str.size());
    exit(1);
  }
  if (write(1, buffer, taken) == -1) {
    auto errno_str = std::to_string(errno);
    write(2, errno_str.data(), errno_str.size());
    exit(1);
  }
}
```

```
errno -l - посмотреть все возможные коды ошибок
```

Раздел 2

9.

Что такое файловый дескриптор?

> **Файловый дескриптор** — это неотрицательное число, которое является идентификатором потока ввода-вывода. Дескриптор может быть связан с файлом, каталогом, сокетом.

Стандартные файловые дескрипторы
0. поток ввода

1. поток вывода
2. поток ошибок

Расскажите про сисколлы open, close и lseek для работы с файлами.

open

> int open(const char *pathname, int flags [, mode_t mode])*
> *int creat(const char* pathname, mode_t mode)

- есть openat и openat2 - принимают дополнительный аргумент int dirfd
- можно получить новый файловый дескриптор при помощи open

```
int main() {
  int fd = open("./input.txt", O_RDONLY);
  char buff[100];
  read(fd, buf, 10);
  write(1, buf, 10);
}
```

close

> int close(int fd)

lseek

> off_t lseek(int fd, off_t offeset, int whence)

- задает отступ при чтении из файла/запись

Что произойдет, если сделать lseek на позицию больше чем размер файла и записать туда что-либо?

```
int main() {
  int fd = open("./output.txt", O_WRONLY|O_CREAT);
  int offset = lseek(fd, 50, SEEK_SET);
  const char* buf = "Hello world";
  int res = write(fd, buf, 10);
  std::cout << res;
}
```

```
cat output.txt #Hello worl
# размер output.txt 60 байт. Реально занимает 10
cp output.txt output2.txt
# пустота заполнится нулями, размер output2.txt - 60 и реально занимает 60
```

10.

Перенаправление ввода-вывода. Как в терминале направить вывод команды в файл с перезаписью файла? А без перезаписи, в режиме добавления в файл?

```
./a.out > result.txt # с перезаписью
./a.out >> result.txt # без перезаписи
```

Как перенаправить вывод одного из потоков (cout или cerr) в файл или в другой поток?

```
./a.out 1>output.txt
./a.out 1>&2
```

Как подавить вывод какого-то из потоков?

```
./a.out 1> /dev/null
```

Что делает команда tee?

tee дублирует вывод: записывает в поток вывода и в еще один указанный источник

```
./a.out | tee result.txt
```

Что делают сисколлы dup и dup2?

> int dup(int oldfd)
> int dup2(int oldfd, int newfd)

- возвращает новый файловый дескриптор, который будет указывать на то же самое, на что указывал старый
- занимает наименьший незанятый
- dup2 использует номер, который указан в newfd. Если newfd открыт, то он закрывается
  Пример: открыли файл, закрыли поток ввода (0), вызвали dup от 3. Теперь 0 указывает на то же самое, на что и 3

11.

Файловые системы. Что такое файловая система? Перечислите, какие наиболее популярные файловые системы существуют на сегодняшний день

> Файловая система — порядок, определяющий способ организации, хранения и именования данных на носителях информации. Информация, какая ФС на вашем жёстком диске записана где-то на нём. ОС знает где это и исходя из этих данных строит дерево файлов.

```
df -T -h # посмотреть тип своей файловой системы
```

Наиболее популярные файловые системы:

- Windows – **FAT**, NTFS, exFAT

- macOS – HFS, APFS, HFS+
- Linux – EXT2, EXT3, EXT4, XFS, JFS

Какие 6 видов файлов существуют в Linux?

1. обычные файлы
2. директории
3. символические ссылки
4. потоки (fifo)
5. сокеты
6. устройства (b/c) - блочные или символьные

Что из себя представляют директории с точки зрения файловой системы?

Директории - особый вид файлов, которые не занимают место на диске и имеют два имени: . и имя папки

Что такое inode и как узнать inode для файлов в данной директории?

inode - уникальный идентификатор для каждого файла. Благодаря inode файлы индексируются на диске.

```
ls -li
```

Что такое виртуальная файловая система? Покажите примеры файлов, которым не соответствует никакое дисковое пространство.

Виртуальная файловая система (VFS) в Linux — это абстрактный слой, который объединяет разные файловые системы в единое дерево каталогов и предоставляет универсальный интерфейс для работы с файлами, независимо от их типа (локальные, сетевые, виртуальные).
Примеры: большинство папок в корневой директории содержат виртуальные файлы, не занимающие пространство. Например (практически) все файлы в /proc, /dev/zero, /dev/null, /dev/random

Что такое swapfile?

В операционной системе Linux swapfile служит для временного хранения данных, которые не помещаются в RAM. Эти данные автоматически выгружаются из оперативной памяти в swapfile, чтобы освободить место для более приоритетных процессов.

12.

Как работает и какие сисколлы использует программа mv?

mv создает новое имя для inod'ы и забывает старое. Можно реализовать через сисколлы **link** и потом **unlink** или **rename**

Как работает и какие сисколлы использует программа rm?

rm работает противоположно ln, то есть забывает одно из имен для inod'ы.
Если останется 0 имен, то данные стираются с диска.
Использует сисколл **unlink**

13.

Жесткие и символические ссылки. В чем разница?

Особенности работы с **символическими** ссылками:

- при удалении/перемещении родительского файла становится битой
- при перемещении самой ссылки становится битой
- аналогично работе указателя в C++
  Особенности работы с **жесткими** ссылками:
- "другое название того же самого файла", аналогично работе ссылок в C++
- не может побиться

Как создать жесткую ссылку, символическую ссылку, что они представляют из себя с точки зрения файловой системы?

```
ln -s result.txt a.txt # символическая ссылка
ln result.txt a.txt # hard link - создаем новое
```

Символическая ссылка - особый тип файла, хранящий относительный путь к файлу, на который она ссылается
Жесткая ссылка - новое имя для той же самой inode

Как работает и какие сисколлы использует программа ln в случае создания жестких ссылок и символических ссылок?

Для символических ссылок вызывается **symlink**
Для жестких - **link**

14.

Права доступа к файлам. Как посмотреть, как изменить права доступа?

```
ls -l out.txt # посмотреть права доступа
chmod +x out.txt
можно задать права по
1. битовой маске
2. +x, +w, +r меняет права всем (user, group и other)
3. g+x меняет права для группы
```

Почему r и x - это разные права? Как понимать эти права доступа для директорий?

r - право на чтение, безопаснее чем право на выполнение
x - загрузить бинарный код как инструкцию к процессору
Для директорий:
r - позволяет вывести содержимое директории. Не запрещает оперировать с файлами, лежащими в директории, если известно имя
x - позволяет перейти в директорию. Без права на выполнение нельзя обращаться к файлам, лежащим в папке

Как поменять владельца файла, как поменять группу владельца?

```
chown root out.txt # поменять владельца файла
chgrp root out.txt # поменять группу
```

Какие сисколлы используются для всего вышеперечисленного?

Для chown и chgrp - chown
Для chmod - chmod

fvjh

SUID-бит (Set User ID) — это специальный бит в правах доступа файла в Linux/Unix, который позволяет запускать файл с правами владельца файла, а не того пользователя, который его выполняет. Это важно для выполнения задач, требующих повышенных привилегий.

1. s маленькая указывает, что SUID-бит установлен и файл **является исполняемым**
2. S большая - SUID-бит установлен, но файл **не является исполняемым** (т.е. отсутствует бит `x`). Чаще является ошибкой настройки.
   Примеры:

- /usr/bin/sudo, создавался суперпользователем и выполняется от лица root
- /usr/bin/passwd. Когда обычный пользователь запускает `passwd`, программа выполняется с правами `root`, чтобы иметь доступ к файлу `/etc/shadow`, где хранятся пароли.

```
chmod +s out.txt
```

15.

Как из терминала посмотреть, какие файловые дескрипторы сейчас открыты у данного процесса и какие файлы им соответствуют?

```
cd /proc/"proccess_id"/fd
ls -l
```

Как из терминала посмотреть, какие процессы сейчас держат открытым данный файл?

```
cat < /dev/zero
lsof /dev/zero
```

Как реализовать это на Си с помощью сисколлов (можно не приводить реализиацию полностью, только объяснить идею)?

1.

Реализовать на С можно при помощи сисколов opendir /proc/"proccess_id"/fd. Потом обойти рекурсивно при помощи readdir. Подробные данные выводим благодаря stat

2.

Можно обойти директории всех процессов /proc/"proccess_id"/fd и найти среди всех открытых файловых дескрипторов нужный файл (переход по символьным ссылкам

осуществляется при помощи readlink)

16.

Блочные и символьные устройства. В чем разница? Приведите примеры того и другого.

- **Блочные** - необходимо записывать информацию блоками.
  Пример: /dev/nvme0n1 - жесткий диск
- **Символьные** обеспечивают непрерывный поток байт.
  Пример: /dev/zero - непрерывный поток нулей

Как прочитать данные с какого-нибудь символьного устройства, а также отправить данные на устройство?

Чтобы отправить данные на устройство или считать, нужно использовать обычные open, close, read, write, то есть обращаться с файлами устройств как с обычными. Это поддерживается механизмом виртуальной файловой системы.

Какие в Linux существуют виртуальные устройства и для чего они нужны?

> В Linux виртуальные устройства — это абстракции, которые представляют собой логические устройства, не связанные напрямую с физическим оборудованием.

- /dev/null - дыра. Используется для отбрасывания ненужного вывода
- /dev/zero - поток нулевых байт. Используется для инициализации памяти, создания пустых файлов
- /dev/random и /dev/urandom - поток рандомных байт
- аудио и видео устройства
- сетевые устройства

Что такое монтирование файловых систем?

> **Монтирование файловых систем** — это процесс подключения файловой системы (например, раздела жёсткого диска, флешки, сетевого ресурса или ISO-образа) к дереву каталогов операционной системы, чтобы сделать её содержимое доступным для использования.

В Linux (и других UNIX-подобных системах) дерево каталогов начинается с корня `/`, и каждая смонтированная файловая система становится частью этой структуры.

Как подмонтировать диск в файловую систему и как отмонтировать? Какие команды в терминале и какие сисколлы для этого используются?

```
df # позволяет посмотреть путь к устройству. /dev/sda1 например
sudo umount "изначальная точка" # сначала отмонтируем
sudo mount "путь к устройству" "точка монтирования"
```

Используются соответствующие сисколлы

Раздел 3

17.

Виртуальная память. Что это такое и зачем нужно?

> **Виртуальная память** — это механизм управления памятью в современных операционных системах. Она создаёт абстракцию, при которой каждый процесс видит своё изолированное адресное пространство, а ОС отвечает за преобразование виртуальных адресов в физические.

Зачем это нужно

1. **Изоляция процессов**:
   Каждый процесс работает в своём виртуальном адресном пространстве, что предотвращает доступ к памяти других процессов.
2. **Разделение и защита памяти**:
   Общие данные (например, библиотеки) могут быть мапированы в адресное пространство нескольких процессов, оставаясь защищёнными от несанкционированной записи.
3. **Эффективное использование памяти**

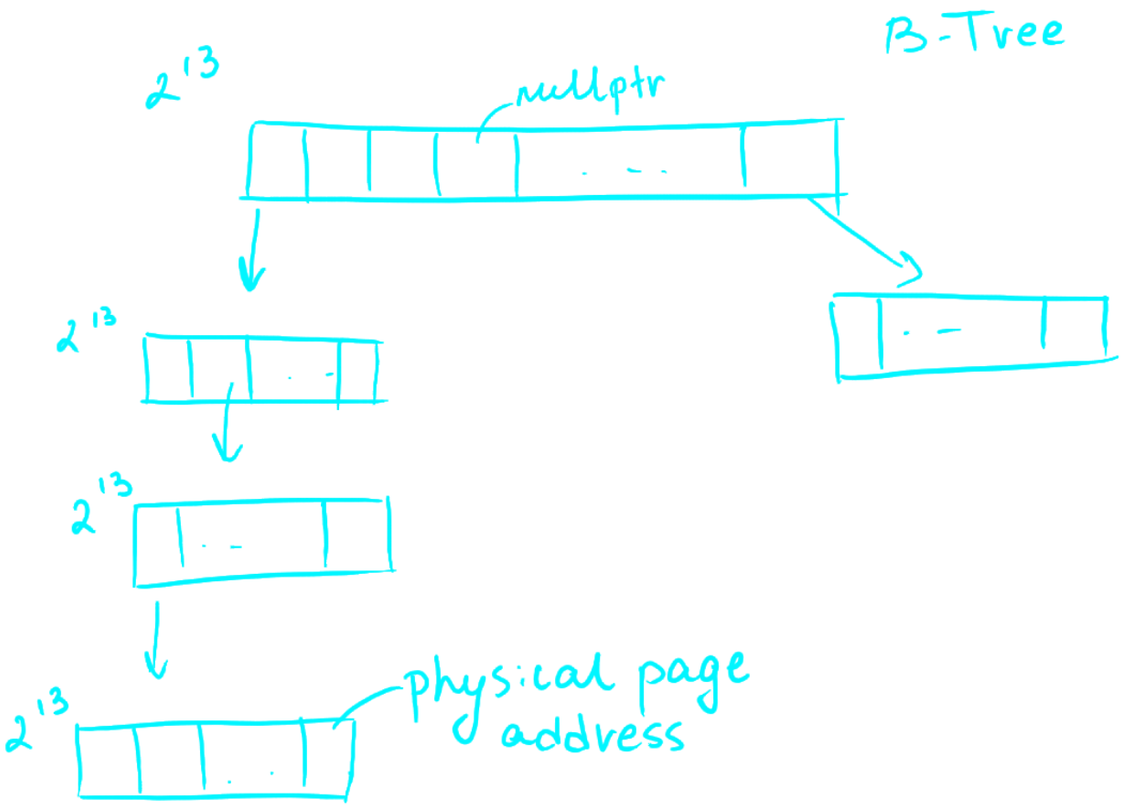
Что такое страничная организация памяти, таблицы страниц, как они устроены и где хранятся?

> **Страничная организация памяти** — это способ управления виртуальной памятью, при котором виртуальное адресное пространство делится на равные блоки (страницы), а физическая память — на блоки той же длины. Размер страницы обычно составляет 4 КБ

> **Таблица страниц** — это структура данных, которая хранит информацию о том, какой физический адрес соответствует каждому виртуальному адресу. При доступе к памяти процессор использует эту таблицу для преобразования адресов.

Таблица страниц хранится в RAM и имеет иерархическую структуру
Адрес страницы (52 бита) разбивается на несколько кусков. Количество кусков = количество уровней в B-дереве. На картинке 4 куска. На нижнем уровне хранятся физические адреса (или ничего, если адреса нет в отображении)

Также хранит права доступа



Что такое page fault, в чем отличие minor от major page fault?

Если при page walk не нашелся физический адрес и в таблице страниц, и в физической памяти, происходит page fault - исключение, обрабатываемое ядром ОС.

1. Minor page fault

- **Описание**:
  Событие, возникающее, когда процесс запрашивает страницу виртуальной памяти, которая **уже находится в физической памяти**, но ещё не связана с таблицей страниц текущего процесса.
  В этом случае ОС обновляет таблицу страниц, чтобы указать, что страница доступна.
- **Пример ситуации**:
  - При создании нового процесса через `fork()` память родительского процесса не копируется сразу. Вместо этого дочерний процесс использует те же страницы в режиме только для чтения.
  - Когда один из процессов пытается изменить данные, возникает minor page fault, и ОС копирует страницу для сохранения независимости данных.

Возникает по причине того, что ОС реализует **ленивую загрузку** (lazy loading), чтобы не выделять ресурсы заранее. Например, при запуске программы ОС может не связывать все виртуальные страницы с физической памятью, а делать это только при первом обращении.

2. Major page fault

- **Описание**:
  Происходит, если запрошенная страница отсутствует не только в таблице страниц, но и в физической памяти. Тогда ОС должна загрузить страницу с диска (например, из файла подкачки или с исполняемого файла на диске) в оперативную память.

- **Пример ситуации**:
  Процесс обращается к данным, которые давно не использовались и были выгружены в файл подкачки (swap).

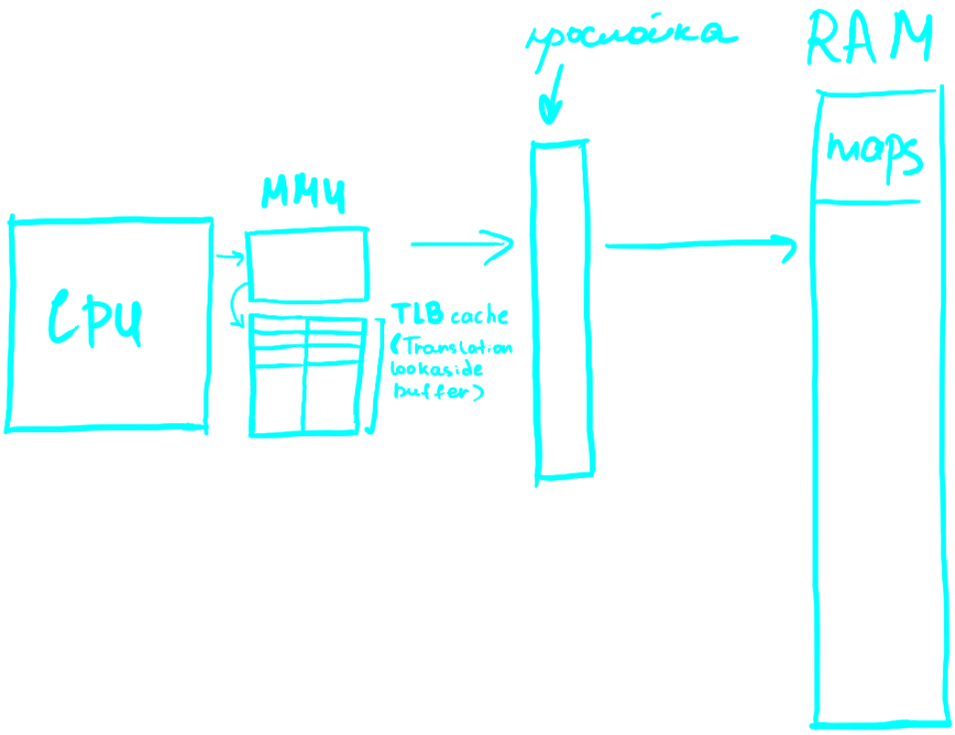
Что такое TLB cache?

> TLB cache - специальный кэш процессора, который хранит недавно использованные соответствия виртуальных и физических адресов.

Как происходит обращение процессора по адресу к памяти с учетом всего вышеназванного (вопрос с открытым ответом)?

Memory management unit выполняет следующие действия:

1. Процессор пытается обратиться по виртуальному адресу
2. Если виртуальный адрес попал в нетранслируемую область, то физический адрес приравнивается к виртуальному
3. Иначе MMU пытается найти в TLB cahce запрашиваемый адрес.
   Если нашелся, то берет физический адрес из отображения
4. Иначе происходит page walk по структуре maps: если физический адрес нашелся, он загружается в TLB cache, иначе страница ищется на диске и подгружается в оперативную память
5. Проверяются права доступа. Если нет нужных прав, то segfault
6. Если page walk завершился неудачно, то page fault: процесс прерывается, ОС обрабатывает ошибку.
   Если память была запрошена, но не выделена (так как она выделяется лениво), то выдается новая страница. Иначе segfault



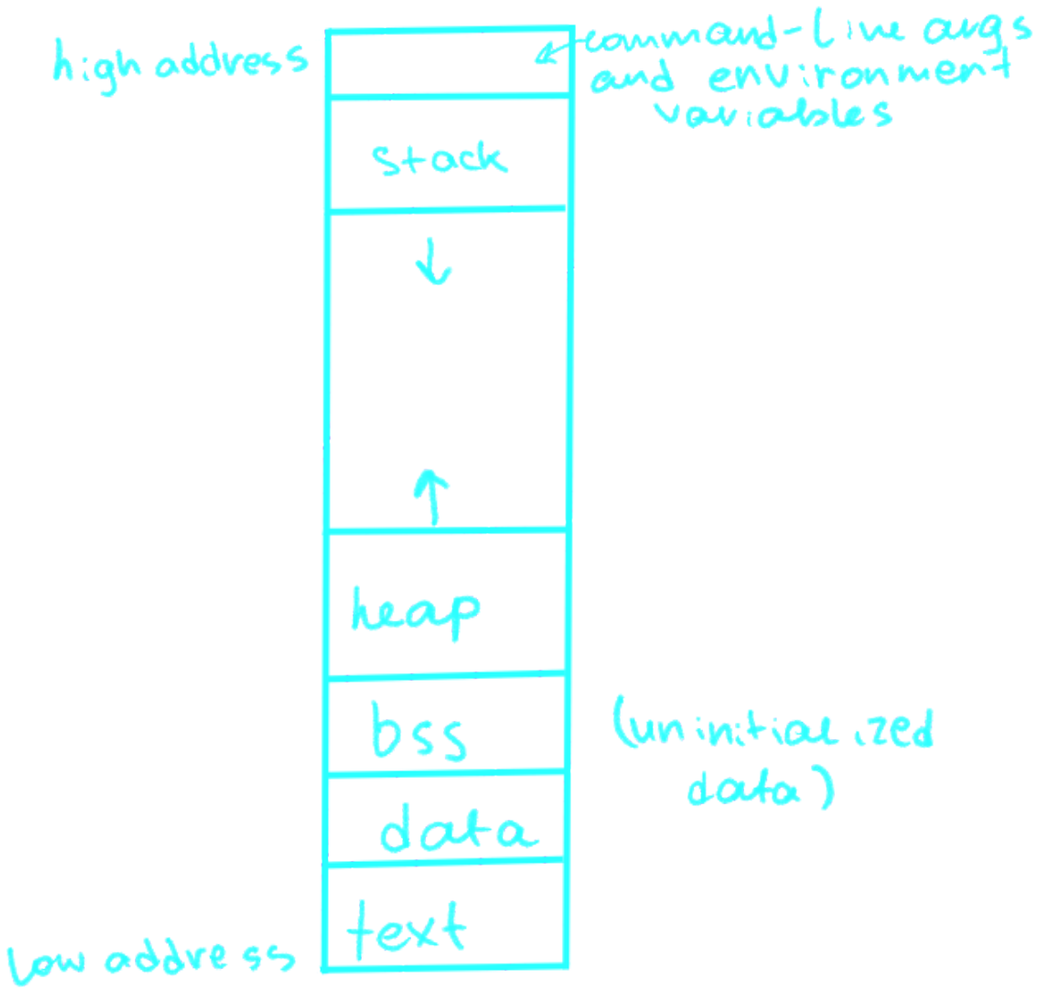
В какой ситуации возникает ошибка Segmentation fault и что в этой ситуации происходит на уровне ОС и процессора?

При обработке исключения page fault ядром ОС может сгенерироваться ошибка segfault, если страница не будет найдена (ОС не обещала выдать запрашиваемую страницу) или

на нее будут выставлены не соответствующие права доступа. После этого процесс аварийно завершает свою работу.

18.

На какие секции делится адресное пространство процесса?



В чем разница между секциями .data, .rodata и .bss?

- **data** - проинициализированные статические переменные
- **rodata** - read only статические переменные
- **bss** - непроинициализированные статические переменные

Зачем нужны сисколлы brk и sbrk?

```
cat /proc/$$/maps
 # 5f60b9861000-5f60b9a0d000 rw-p 00000000 00:00 0
[heap]
 # 5f60b9a0d000 - program break
```

brk возвращает адрес, который в данный момент является program break.
sbrk позволяет отодвинуть границу program break'a. Увеличение - выделение памяти процессу, уменьшение - деаллокация. В случае успеха возвращает границу, которая была до изменения

Как посмотреть, как выглядит в данный момент адресное пространство процесса?

```
cat /proc/$(pgrep a.out)/maps
```

19.

Можно ли запросить конкретный виртуальный адрес для выделения памяти?

Да, нужно передать желаемый адрес в качестве первого параметра. Не гарантируется, что в итоге память выделится конкретно на этом адресе.

Почему при обращении за границу массива segfault происходит не всегда?

Если обращаемся в пределах памяти, которая была выделена программе (минимум одна страница, то есть 4Кб) и не нарушаются никакие права доступа,то segfault не происходит

Как с помощью mmap загрузить файл в оперативную память? Можно ли таким образом поменять файл? Зачем нужен сисколл msync?

```
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <iostream>

int main() {
  int fd = open("example.txt", O_RDONLY);
  if (fd == -1) {
    exit(1);
  }

  struct stat sb;
  if (fstat(fd, &sb) == -1) {
    exit(1);
  }

  char* data = (char*)mmap(NULL, sb.st_size, PROT_READ | PROT_WRITE,
MAP_PRIVATE, fd, 0);
  if (data == MAP_FAILED) {
    exit(1);
  }

  data[0] = 'X';
  std::cout << data << std::endl;

  if (msync(data, sb.st_size, MS_SYNC) == -1) {
    exit(1);
  }

  if (munmap(data, sb.st_size) == -1) {
    exit(1);
  }
}
```

MAP_PRIVATE: создается локальная копия данных, и ее изменение не повлияет на содержимое файла. Можно менять data даже с O_RDONLY
MAP_SHARED: можно менять файл при наличии нужных прав.

Нет гарантий, что файл поменяется сразу, так как операция ассинхронная. Поэтому**msync** синхронизирует изменения, чтобы до закрытия файла они гарантированно произошли.

20.

Какие бывают права доступа к памяти?

PROT_READ, PROT_WRITE, PROT_EXEC - права на чтение, запись, выполнение

Зачем нужен сисколл mprotect и как им пользоваться?

mprotect - сисколл, позволяющий изменить права доступа к памяти.
Адрес должен быть выровнен по размеру страницы.

> int mprotect(void* address, size_t len, int prot)

Покажите, как с помощью mmap и mprotect загрузить код из библиотеки в память на выполнение.

```
#include <sys/stat.h>
#include <unistd.h>
#include <iostream>
#include <fcntl.h>
#include <sys/mman.h>

int main(int argc, char* argv[]) {
  const char* file_name = argv[1];
  double argument = strtod(argv[2], NULL);

  int fd = open(file_name, O_RDONLY);
  struct stat st = {};
  fstat(fd, &st);

  void* addr = mmap(NULL, st.st_size, PROT_READ|PROT_EXEC, MAP_PRIVATE, fd,
0);
  double (*func)(double) = (double (*)(double)) ((char*)addr + 0x40);

  close(fd);
  double result = func(argument);
  std::cout << argument << std::endl << result;
  munmap(addr, st.st_size);
}
```

нужен файл с динамической библиотекой

```
double mysqr(double x) {
  return x * x;
}
```

```
g++ -c func.c
readelf -a func.o # смотрим на колонку align для секции .text
```

```
g++ mmap_library.cpp
./a.out func.o 5
```

Что означает ошибка Illegal intstruction?

Illegal intstruction - процессор начал выполнять бинарный код как ассемблерные инструкции, но при раскодировке увидел невалидные команды

21.

Как реализованы функции malloc и free в стандартной библиотеке Си? Расскажите про механизм бакетов, малые и большие бакеты. Как malloc выбирает, какой бакет использовать? Как происходит освобождение бакетов и слияние соседних свободных бакетов?

- malloc пытается минимизировать количество вызовов сисколлов sbrk и mmap
- в **MAP_THRESHOLD** записано число байтов, начиная с которого malloc **начинает использовать mmap**. Где-то используется 128 Кб
- записывает 16 байт на начало выделяемой памяти - размер выделенного куска, чтобы корректно удалить в будущем
- В самом начале мы делаем sbrk на какое-то большое число байт (например на MMAP_THRESHOLD) и тем самым обеспечиваем себе "бассейн"с уже выделенной памятью для использования в дальнейшем (между старым program break и новым). Если место заканчивается, то опять сдвигаем program break (например на 1 страницу), если это возможно.
- Храним **массив из двусвязных списков свободных чанков** размеров от 16 до 512 с интервалом 8 (small chunks), а после размеры чанков будут расти уже экспоненциально, пока не дорастут до MMAP_THRESHOLD (large chunks)
- Каждый свободный чанк лежит в двусвязном списке, хранит свой размер **в первых и последних** 4 байтах. Также хранит указатели на предыдущий свободный чанк из списка и следующий для удаления из списка за O(1).
- Каждый занятый чанк хранит свой размер в начале и конце. Посередине - занятое программой, попросившей выделить память, место
- Когда просят выделить немного памяти, то идем в структуру выше и пытаемся найти свободный блок, в который влезет выделяемое содержимое (то есть по размеру минимальный больше равный). Если не находим, то откусываем от нашего хранилища блок того размера, которого мы не смогли найти в свободных чанках. Помимо этого запомним, что теперь наше хранилище начинается позже, так как его начало ушло на новый чанк.
- Когда освобождается чанк в этом хранилище, то информация о размерах, записанная в конце, позволяет объединять чанк со своими соседями, перед тем, как возвращать его в список свободных чанков (размер кратен 8, а значит последний бит размера можно использовать для индицирования занятости блока). Чанк добавляется в массив

Что такое fastbins?

fastbins – только что освобождённые small chunks. Утверждается, что если вы освободили кусок какого-то размера, то скорее всего вы скоро попросите выделить кусок такого же размера.
После удаления и перед тем как добавить в массив, их связывают в отдельный список. И когда запрашивают сколько-то байт, то сначала проверяются чанки из fastbins. Если там

не удалось найти подходящего (best suit) чанка, то fastbins очищается и чанки раскидываются по алгоритму по своим бакетам в массив.

Какие сисколлы использует malloc для работы с памятью?

malloc использует в основном 2 сисколла – brk (sbrk) для небольших размеров и mmap для значительных запросов.

Раздел 4

22.

Что такое процесс?

> Процесс - запущенный исполняемый код, имеющий аттрибуты, которые навесила на него операционная система, некоторая сущность в ядре ОС.

Как посмотреть все процессы в системе? Что такое pid, ppid? Как посмотреть дерево процессов? Как посмотреть потребление памяти, потребление CPU каждым из процессов?

```
ps aux # вывести все процессы
pstree # вывести дерево процессов, или F5 в htop

htop # можно посмотреть список всех процессов с помощью утилиты htop
top # стандартная команда
```

VIRT - количество выделенной виртуальной памяти, **RES** - фактически занятая память
SHR - занятая разделяемая память (shared memory)
CPU - сколько процесс отъедает от ядра процессора

PID - id процесса
PPID - id родительского процесса
Можно в htop включить отображение PPID

Что такое приоритет процесса, какой он бывает, как его узнать и как поменять?

Посмотреть приоритет можно в htop - колонка **NI** (niceness).
Niceness - сколько процесс готов "уступить" процессорного времени. Чем больше NI, тем ниже приоритет. Варьируется от -20 до 20, по умолчанию 0.
Priority (PR) высчитывается на основе niceness для обычных процессов.

```
nice -n 10 "command" # позволяет запустить процесс с определенным NI (10)
# для отрицательного niceness нужны права root'a
renice -n 10 "command" # изменить NI у работающего процесса на 10
```

Что такое uid, euid и cwd данного процесса, как их узнать и как поменять?

> **uid** - id **настоящего** юзера, который запустил процесс. Определяет базовые права доступа (например, какие файлы или ресурсы может использовать процесс).
> **euid** - id юзера, от чьего имени был запущен процесс.
> `Пример` : вызов sudo. У вызванной программы uid оказывается пользователя, вызвавшего sudo, а euid - id root'a

> **cwd** - это **текущая рабочая директория процесса**, относительно которой выполняются операции с файлами.

Как узнать?

Можно при помощи htop или ps -eo pid,user,euser,cmd

Как изменить?

- cwd

```
#include <unistd.h>
int main() {
    chdir("/new/directory/path");
    return 0;
}
```

- uid/euid

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

int main() {
    if (setuid(1001) != 0) { // seteuid
        perror("Failed to set UID");
        return 1;
    }
    return 0;
}
```

Можно изменить для работающих процессов при помощи gdb

23.

Расскажите про сисколлы fork и exec. Какие версии сисколла exec существуют и в чем разница между ними?

fork - создает новый процесс, дублируя процесс, который вызвал сисколл. За исключением PID старый и новый процесс неотличимы на момент создания. Новому созданному процессу вернется 0, а старому - PID нового процесса.

```
// вопрос на уд на фпми
#include <iostream>
#include <unistd.h>

int main() {
  std::cout << "Hello!" << "\n";
  int pid = fork();
  if (pid == 0) {
    std::cout << "child" << "\n";
  } else {
    std::cout << "parent" << "\n";
  }
}
// вывод: 2 раза Hello, child, parent, так как буфер cout тоже копируется
```

exec - не сисколл если на душном. Вместо текущего процесса открывает новую программу, переданную в качестве первого аргумента, сохраняет PID старого процесса, файловые дескрипторы. Куча и стек не сохраняются

Существует несколько вариантов `exec`, которые предоставляют разные способы передачи аргументов и переменных окружения новому процессу. Все они являются оболочками вокруг одного системного вызова `execve`.

1. **execl** - передача аргументов как отдельных строк
   `int execl(const char *path, const char *arg, ..., NULL);`
   `execl("/bin/ls", "ls", "-l", NULL);`
2. **execv** - передача массива строк
   `int execv(const char *path, char *const argv[]);`

   ```
   char *args[] = {"ls", "-l", NULL};
   execv("/bin/ls", args);`
   ```

3. **execvp** - аналог execve, но ищет исполняемый файл в системной переменной PATH
   `int execvp(const char *file, char *const argv[]);`

```
      char *args[] = {"ls", "-l", NULL};
      execvp("ls", args);
```

И так далее, все разновидности в man 3 exec.

Функция	Передача аргументов	Передача переменных окружения	Поиск по `PATH`
`execl`	Отдельные аргументы	Нет	Нет
`execv`	Массив строк	Нет	Нет
`execle`	Отдельные аргументы	Да	Нет
`execve`	Массив строк	Да	Нет
`execlp`	Отдельные аргументы	Нет	Да
`execvp`	Массив строк	Нет	Да

Можно заметить, что каждая буква за что-то отвечает. p - за path, l - за передачу аргументов в качестве отдельных строк, v - передача аргументов в качестве массива, e - передача переменных окружения.

```
which ls # посмотреть по какому адресу находится исполняемая программа
# нужно для передачи в качестве первого аргумента execve
```

В чем необычность функций fork и exec, что происходит при их вызове?

хз если честно какой ответ тут ожидается, но попросить дедушку мороза (процессор) проснуться в теле другого человека, или проснуться в одном мире со своей копией (привет субстанция) ну странновато да..... тут ответ гптшки

`fork`:

- Уникальная концепция: Два независимых процесса начинают выполнение с одной точки.
- Сохранение ресурсов: Благодаря Copy-on-Write, `fork` эффективен даже для процессов с большим объёмом памяти.

`exec`:

- Замена без изменения PID: Содержание процесса полностью заменяется, но процесс остаётся тем же.
- Простота интеграции: Сохраняются открытые дескрипторы и PID, что упрощает коммуникацию между процессами.

Покажите пример вызова из программы другой программы, используя fork+exec.

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

int main() {
  int pid = fork();
  if (pid != 0) {
    std::cout << "parent" << std::endl;
    char* argv[] = {"/usr/bin/ls", "/home/liza", NULL};
    char* envp[] = {NULL};
    int code = execve("/usr/bin/ls", argv, envp);
    std::cout << "code: " << code << "\n";
    std::cout << errno << "\n";
  } else {
    std::cout << "child" << "\n";
  }
}
```

Что такое fork-бомба?

Fork-bomb — это программа, которая бесконтрольно создаёт новые процессы через системный вызов `fork()`, пока система не исчерпает ресурсы (например, процессорное время или максимальное количество процессов). Это вызывает нарушение работы системы или её полное зависание.

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

int main() {
  while (true) {
        int pid = fork();
  }
}
```

24.

Какие бывают состояния у процессов?

man ps перечислит

- **S** - прерываемый сон, то есть ждет события, чтобы проснуться. Пример: ввод с консоли

- **D** - непрерываемый сон, выполняет критическую операцию ядра (обычно связанную с I/O)
- **T** - остановленный процесс (SIGSTOP или Ctrl+z)
- **t** - процесс остановлен отладчиком
- **Z** - зомби-процесс
- **R** - процесс выполняется или готов к выполнению

Как в терминале приостановить процесс, как возобновить приостановленный процесс?

Если процесс запущен в терминале, то Ctrl+z, иначе:

```
kill -STOP "proccess-id"
kill -CONT "proccess-id" # у меня не работает))))))))))))))))))))))))))))0
# оказалось если процесс ожидает ввода с терминала (ну в целом находится в состоянии S+, то ему может быть трудно проснуться)
# например если процесс ждал ввод, то он может не просыпаться до того момента, пока этот ввод не получит
```

Как пользоваться командами fg и bg?

```
./a.out & # запустить программу на фоне
jobs # посмотреть список фоновых процессов, запущенных в данном терминале
```

fg

Выводит на передний план последнюю job, запущенную в терминале

bg

Запускает в фоновом режиме приостановленную задачу. Можно использовать синтаксис bg 1 - запустить определенную задачу

Что такое процессы-зомби, как они возникают?

Процессы-зомби - процессы, родители которых завершились, но родитель еще не собрал информацию о них.

- Это происходит, если родительский процесс не вызвал `wait()` для получения статуса завершённого процесса.
- Зомби-процесс не занимает ресурсов, кроме записи в таблице процессов.

Как посмотреть, в каком состоянии находится сейчас какой-либо процесс?

```
htop # можно найти нужный процесс, колонка S
ps u -p "proccess-id" # колонка STAT
```

25.

Что такое rlimit для процесса?

> rlimit - ограничение на потребление процессором различных ресурсов. Текущие лимита процесса можно посмотреть в файле командой cat /proc/$(pgrep a.out)/limits

Как пользоваться функциями getrlimit и setrlimit?

Сиколлы используют структуру

```
struct rlimit {
    rlim_t rlim_cur;  // Текущее ограничение ресурса
    rlim_t rlim_max;  // Максимально возможное ограничение ресурса
};
```

```
int getrlimit(int resource, struct rlimit *rlim);
int setrlimit(int resource, const struct rlimit *rlim);
```

resource задается константой - ресурс, лимит которого хотим узнать. Смотрим man 2 getrlimit. В структуре меняем меняем/смотрим нужное нам значение.

Как установить процессу ограничение на использование памяти и/или процессорного времени?

Можно через setrlimit установить ограничение на **RLIMIT_DATA** - вся память в совокупности, считая стек, кучу и тд, и на **RLIMIT_CPU** - время исполнения в секундах.

Что произойдет, если эти ограничения будут превышены?

Превышение лимитов ресурсов в системе приводит к различным ошибкам, которые можно обработать в программе, но в случае критических лимитов, таких как время процессора, операционная система может принудительно завершить процесс.

Если же пытаться выделить больше памяти, чем можно, или открыть больше файлов, то сисколлы будут возвращать ошибку и менять errno.

26.

Расскажите про библиотеку seccomp.

> **Seccomp (Secure Computing)** — это механизм в Linux, который позволяет ограничивать процесс в использовании определённых системных вызовов

```
Если процесс вызывает запрещённый системный вызов, происходит событие
(например, `SIGSYS`), и программа завершается с ошибкой.
```

Покажите на примере, как запретить программе вызывать определенные сисколлы. Как получить ошибку Bad system call (core dumped)?

```
sudo apt-get install libseccomp-dev # установка библиотеки
g++ seccomp.cpp -lseccomp # линкуем либу
```

```
#include <seccomp.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <iostream>

void seccomp_setup() {
```

```
  scmp_filter_ctx ctx = seccomp_init(SCMP_ACT_ALLOW);
  if (ctx == nullptr) {
    std::cout << "seccomp_init failed" << std::endl;
    exit(1);
  }

  if (seccomp_rule_add(ctx, SCMP_ACT_KILL, SCMP_SYS(clone), 0) != 0) {
    std::cout << "seccomp_rule_add failed" << std::endl;
    seccomp_release(ctx);
    exit(1);
  }

  if (seccomp_load(ctx) != 0) {
    std::cout << "seccomp_load failed" << std::endl;
    seccomp_release(ctx);
    exit(1);
  }
  seccomp_release(ctx);
}

int main() {
  seccomp_setup();
  fork();
  std::cout << errno;
}
```

```
man seccomp_init # подробную информацию можно найти в мануале
```

27.

Что такое сигналы?

> Сигналы - вид IPC (Inter Proccess Communication)

Как послать сигнал процессу из терминала, а также из кода программы?

```
kill -signal "proccess_id"
```

Есть одноименный syscall - kill. Можно использовать из программы, чтобы посылать процессам сигналы

Перечислите известные вам стандартные сигналы с объяснением, для чего они применяются.

- SIGHUP - подключились удаленно через ssh к серверу, запустили вычисления на нем. Если во время этого оборвется соединение, процессу на сервере придет SIGHUP. Программа по умолчанию падает. nohup позволяет продолжить работу
- SIGABRT - сигнал от abort()
- SIGCHLD - посылается родителю, когда ребенок завершился
- SIGCONT - сигнал продолжения работы процесса, если он был остановлен
- SIGILL - illegal instruction
- SIGSYS - bad system call
- SIGSEGV - segfault
- SIGTSTP - Ctrl+z
- SIGSTOP - предыдущей сигнал, только этот нельзя заблокировать

- SIGFPE - floating point exeption
- SIGINT - Ctrl+c
- SIGQUIT - Ctrl + \
- SIGTERM - посылается по умолчанию, если написать kill
- SIGKILL - kill -KILL
  и так далее, больше в man 7 signal
  Только 2 сигнала нельзя перехватить: **SIGKILL и SIGSTOP**

Какова стандартная реакция процессов на каждый из сигналов?

Есть 5 видов стандартной реакции:

- Ign - проигнорировать
- Term - убиться
- Core - убиться и сделать coredump
- Stop - остановиться
- Cont - возобновиться

Как вручную из терминала вызвать у стороннего процесса segfault?

```
kill -11 <pid>
```

Как из кода послать сигнал самому себе?

```
kill(getpid(), SIGTERM) // syscall
raise(SIGTERM) // то же самое
```

Как в коде программы заснуть до прихода сигнала?

```
pause() // syscall
```

28.

Как сделать кастомный обработчик сигналов?

Можно использовать **signal** или **sigaction**
signal - устаревший и предоставляющй мало возможностей метод
sigaction - хороший и крутой.
sigaction позволяет выставить разные настройки. Например SA_NODEFER не блокирует одинаковые сигналы во время их обработки.

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

void handler(int signum) {
  std::cout << "signal occurred\n";
  sleep(5);
}

int main() {
  struct sigaction sa;
```

```
  sa.sa_handler = &handler;
  sigemptyset(&sa.sa_mask);
  sa.sa_flags = SA_NODEFER;

  if (sigaction(SIGINT, &sa, NULL) == -1) {
    std::cout << "sigaction";
    exit(1);
  }

  sleep(5);

  int* ptr = NULL;
  *ptr = 10;
  std::cout << "after segfault";
}
```

Замечание к коду: sleep прерывается либо сигналом, либо истечением таймера, поэтому обратно в сон не уходим после того, как отработает последний сигнал.

Покажите на примере, как из кода программы перехватывать segfault и делать что-то нестандартное при его наступлении.

```
#include <stdio.h>
#include <iostream>
#include <signal.h>

void handler(int signum) {
  std::cout << "caught " << signum << std::endl;
}

int main() {
  signal(SIGSEGV, &handler);
  int* p = nullptr;
  std::cout << *p;
}
```

Будет бесконечно печаться caught 11, так как после того, как сигнал был обработан, мы возвращаемся на то же место программы, где получили сигнал

```
#include <stdio.h>
#include <iostream>
#include <signal.h>

int* p = NULL;
int a = 0;

void handler(int signum) {
  std::cout << "caught " << signum << std::endl;
  p = &a;
}

int main() {
  signal(SIGSEGV, &handler);
  std::cout << *p;
}
// ситуация не исправится так как ассемблерные инструкции не меняются
```

Что происходит, если во время обработки сигнала приходит другой сигнал?

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

void handler(int signum) {
  printf("Signal number %d received\n",#include <stdio.h>
#include <signal.h>

void handler(int signum) {
  printf("Signal number %d received\n", signum);
  sleep(5);
  printf("Signal number %d done\n", signum);
}

int main() {
  signal(SIGINT, &handler);
  signal(SIGTSTP, &handler);
  getchar();
} signum);
  sleep(5);
  printf("Signal number %d done\n", signum);
}

int main() {
  signal(SIGINT, &handler);
  signal(SIGTSTP, &handler);
  getchar();
}
```

- Если обрабатывали SIGINT, но пришел SIGTSTP, то начнем обрабатывать SIGTSTP
- Если послать сигнал, который уже обрабатывается, то он начнет обрабатываться после того, как предыдущий закончит. Т.е. нажали Ctrl+c, крутимся в обработчике, нажали еще раз Ctrl+c - ничего не происходит. Как выйдем из handler, начнем заново обрабатывать SIGINT (но только один разэ, даже если пришло 10 сигналов SIGINT во время обработки)
- Если прилетел сегфолт при обработке сегфолта, вызовется стандартный обработчик

Как можно заблокировать получение других сигналов во время обработки какого-либо сигнала?

Можно использовать sigaction.

```
sigfillset(&sa.sa_mask); // такая строчка заблокирует все приходящие сигналы
sigaddset(&sa.sa_mask, SIGUSR1); // или можно заблокировать только некоторые
sigaction(SIGINT, &sa, NULL)
```

Что происходит, если сигнал приходит во время выполнения какого-либо сисколла?

1. Системный вызов прервывается
   Для многих системных вызовов возвращается ошибка, и глобальная переменная `errno` устанавливается в значение `EINTR` (Interrupted System Call).

2. Системный вызов возобновляется (automatically restarted)
   Для некоторых системных вызовов можно настроить автоматическое возобновление после обработки сигнала, если установить флаг `SA_RESTART` в структуре `sigaction`.
3. Системный вызов не прерывается
   Пример: вызовы, работающие с флагом `O_NONBLOCK` (неблокирующий ввод-вывод).

29.

Что такое pipes?

> Pipes - второй вид IPC.

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

- принимает два указателя на инт - в них записывает два файловых дескриптора, из первого можно читать, во второй - писать
- Позволяет общаться либо с собственным ребенком, либо общаться между детьми

Покажите в коде пример создания pipe и общения между двумя процессами с помощью pipe.

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

int main(int argc, char* argv[]) {
  char buf[1000];
  int pipefd[2];
  pipe(pipefd);

  int cpid = fork();
  if (cpid == 0) {
    close(pipefd[1]);

    while (read(pipefd[0], &buf, 1) > 0) {
      write(STDOUT_FILENO, &buf, 1);
    }

    write(STDOUT_FILENO, "\n", 1);
    close(pipefd[0]);
    _exit(0);
  } else {
    close(pipefd[0]);
    sleep(1);
    write(pipefd[1], argv[1], strlen(argv[1]));
    close(pipefd[1]);
    wait(NULL);
    exit(0);
  }
}
```

Объяснение: родитель **не сразу** начинает писать в pipe, но while у ребенка не завершается, так как чтение из pipe **блокирующее**. То есть если вызван read, но читать

нечего, то он закончится только если прочитает что-то, или другой конец трубы закроется. В данном случае если не закрыть pipefd[1] в родителе, то зависнем

- если читать из пайпа, в который никто не пишет, возвращается 0

В какой ситуации возникает ошибка Broken pipe?

Broken pipe возникает при **записи в закрытый конец пайпа**

```
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <signal.h>
#include <iostream>

void handle(int signum) {
  std::cout << "caught " << signum << std::endl;
}

int main(int argc, char* argv[]) {
  int pipefd[2];
  pipe(pipefd);
  signal(SIGPIPE, &handle);
  int cpid = fork();
  if (cpid == 0) {
    close(pipefd[1]);
    close(pipefd[0]);
    _exit(0);
  } else {
    close(pipefd[0]);
    sleep(1);
    write(pipefd[1], argv[1], strlen(argv[1]));
    close(pipefd[1]);
    wait(NULL);
    exit(0);
  }
}
```

Как реализовать аналог оператора | в bash на Си?

Реализация аналога оператора `|` (пайплайна) в Bash на языке C подразумевает использование системных вызовов `pipe()`, `fork()`, `dup2()`, и `execvp()`. Оператор `|` передаёт вывод одного процесса как ввод другому, что достигается с помощью создания неименованного канала (pipe) между двумя процессами.

30.

Что такое fifo-файлы?

> FIFO - третий вид IPC.

- Представлены в виде файлов, имеющих тип p (pipe) и не хранящихся на диске
- Буфер `PIPE_BUF` имеет размер порядка нескольких килобайт. Если несчитанных данных будет больше, то программа, пытающаяся писать в fifo файл, зависает до того момента, как данные будут обработаны.

Как создать такой файл из терминала, а также программно?

```
mkfifo myfifo # в терминале
int mkfifo(const char* pathname, mode_t mode) # из кода
```

Что, если несколько процессов пишут в один и тот же fifo? Что, если несколько процессов читают один и тот же fifo?

Две программы пишут в один файл

- Если несколько процессов одновременно пишут данные в FIFO, то данные от разных процессов могут быть смешаны в произвольном порядке. Однако каждая запись будет атомарной, если её размер не превышает `PIPE_BUF` (обычно 4096 байт на современных системах Linux).
- Если размер записи превышает `PIPE_BUF`, данные могут быть частично перемешаны между процессами.
- **Пример:**
  Процесс A пишет "Hello".
  Процесс B пишет "World".
  Читатель может получить `HelloWorld` или `WorldHello`, но строки "Hello" и "World" останутся целыми, если их длина ≤ `PIPE_BUF`.

  #### Две программы читают из одного файла

  - Когда несколько процессов читают данные из одного и того же FIFO, данные разделяются между процессами, так как FIFO работает как очередь. Каждая прочитанная часть данных удаляется из FIFO и становится недоступной для других процессов.
- **Пример:**
  - Если процесс A и процесс B читают из одного FIFO:
    Процесс A может получить первую часть данных.
    Процесс B получит следующую часть.
    Никто из них не получит полные данные одновременно.

31.

Что такое разделяемая память?

> Shared memory - третий вид IPC.

Есть два способа работы с разделяемой памятью - далее 1 и 2.
Первый - использование функций System five, то есть старых линуксовых, которых может не быть в других UNIX-подобных системах.
Второй - использование функций, соответствующих стандарту POSIX.

Какие сисколлы существуют для создания и управления разделяемой памятью?

1.

```
int shmget(key_t key, size_t size, int shmflg) // выделить память
void* shmat(int shmid, const void* shmaddr, int shmflg) // attach,
присоединиться к памяти
int shmdt(const void* shmaddr) // отсоединиться
int shmctl(int shmid, int cmd, struct shmid_ds *buf); // контроль над
```

```
памятью, в том числе удаление
key_t ftok(const char *pathname, int proj_id)
```

2.

```
int shm_open(const char *name, int oflag, mode_t mode); // выделить память
int shm_unlink(const char *name); // удалить
```

ftok - отдает ключ по пути к **существующему** файлу, один из способов не брать рандомный ключ. proj_id - ненулевое число.

Покажите на примере, как устроить общение через разделяемую память между двумя процессами.

1.

```
// writer
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SHM_SIZE 1024  // Размер разделяемой памяти

int main() {
  key_t key = 1234;  // Тот же ключ, что и в writer.c

  // Получение сегмента разделяемой памяти, IPC_CREAT говорит, что нужно
создать память
  int shmid = shmget(key, SHM_SIZE, IPC_CREAT | 0666);
  if (shmid < 0) {
    perror("shmget");
    exit(1);
  }

  // Присоединение сегмента разделяемой памяти
  char *data = (char *)shmat(shmid, NULL, 0);
  if (data == (char *)(-1)) {
    perror("shmat");
    exit(1);
  }

  // Пишем в память из потока ввода
  fgets(data, SHM_SIZE, stdin);
  getchar();

  shmdt(data);

  return 0;
}
```

```
// writer
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

```c
#define SHM_SIZE 1024  // Размер разделяемой памяти

int main() {
  key_t key = 1234;  // Тот же ключ, что и в writer.c

  // Получение сегмента разделяемой памяти
  int shmid = shmget(key, SHM_SIZE, 0666);
  if (shmid < 0) {
    perror("shmget");
    exit(1);
  }

  // Присоединение сегмента разделяемой памяти
  char *data = (char *)shmat(shmid, NULL, 0);
  if (data == (char *)(-1)) {
    perror("shmat");
    exit(1);
  }

  // Чтение из разделяемой памяти
  printf("Прочитано сообщение: %s", data);

  // Отсоединение от разделяемой памяти
  shmdt(data);

  // Удаление сегмента памяти
  shmctl(shmid, IPC_RMID, NULL);

  return 0;
}
```

Пояснения:

- ключ - некоторый int, однозначно идентифицирующий кусок разделяемой памяти в системе. Нужно угадать ключ, который никто не использует
- IPC_CREAT - создать новый сегмент
- запускаем writer, пишем в консоль сообщение. Запускам reader и получаем написанное сообщение. Круто!

2.

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <unistd.h>
#include <string.h>

#define SHM_NAME "/shared_memory_example"  // Имя разделяемой памяти
#define SHM_SIZE 1024                      // Размер памяти

int main() {
  // Создание объекта разделяемой памяти
  int shm_fd = shm_open(SHM_NAME, O_CREAT | O_RDWR, 0666);
  if (shm_fd == -1) {
    perror("shm_open");
    exit(1);
```

```
  }

  // Установка размера объекта разделяемой памяти
  if (ftruncate(shm_fd, SHM_SIZE) == -1) {
    perror("ftruncate");
    exit(1);
  }

  // Отображение разделяемой памяти в адресное пространство процесса
  char* shared_memory = (char*)mmap(NULL, SHM_SIZE, PROT_READ | PROT_WRITE,
MAP_SHARED, shm_fd, 0);
  if (shared_memory == MAP_FAILED) {
    perror("mmap");
    exit(1);
  }

  // Запись в разделяемую память
  fgets(shared_memory, SHM_SIZE, stdin);

  // Очистка ресурсов
  munmap(shared_memory, SHM_SIZE);
  close(shm_fd);

  getchar();
  return 0;
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <unistd.h>

#define SHM_NAME "/shared_memory_example"  // То же имя разделяемой памяти
#define SHM_SIZE 1024                      // Размер памяти

int main() {
    // Открытие существующего объекта разделяемой памяти
    int shm_fd = shm_open(SHM_NAME, O_RDWR, 0666);
    if (shm_fd == -1) {
        perror("shm_open");
        exit(1);
    }

    // Отображение разделяемой памяти в адресное пространство процесса
    char *shared_memory = mmap(NULL, SHM_SIZE, PROT_READ | PROT_WRITE,
MAP_SHARED, shm_fd, 0);
    if (shared_memory == MAP_FAILED) {
        perror("mmap");
        exit(1);
    }

    // Чтение данных из разделяемой памяти
    printf("Прочитано сообщение: %s", shared_memory);

    // Очистка ресурсов
    munmap(shared_memory, SHM_SIZE);
    close(shm_fd);
```

```
    // Удаление объекта разделяемой памяти
    shm_unlink(SHM_NAME);

    return 0;
}
```

Пояснения:

- Вместо ключа используется имя разделяемой памяти, будет храниться в /dev/shm как виртуальный файл

Как посмотреть, какие участки разделяемой памяти существуют в ОС и кто их создал?

```
ipcs -m
strace ipcs # можно увидеть директорию, откуда ipcs берет информацию
cat /proc/sysvipc/shm
```

Как посмотреть, какие страницы разделяемой памяти сейчас использует данный процесс?

```
cd /proc/<proccess_id>/maps/
# найдем память с необычными правами. Вместо последней буквы p (private)
# стоит s (shared)
```

32.

Что такое потоки выполнения (треды, threads, нити)?

> Threads - процессы, которые разделяют общие ресурсы (кроме стека), то есть (как правило) у них совпадает виртуальное пространство

Покажите пример создания и использования thread на C++.

```
#include <thread>
#include <iostream>

void job(char c) {
  for (int i = 0; i < 100000; ++i) {
    std::cout << c;
  }
}

int main() {
  std::thread t(job, 'a'); // в этот момент запустился отдельный thread
  job('b'); // конкурентное использование функции
  t.join();
}
```

Покажите пример параллельной обработки из двух тредов каких-либо данных

```
void job() {
  std::vector<int> v;
```

```
  for (int i = 0; i < 100000; ++i) {
    v.push_back(i);
  }
}

int main() {
  std::thread t(job);
  job();
  t.join();
}
```

- Все окей, так как маллок адаптирован для многопоточного программирования

Что делают методы join и detach? Что происходит, если main завершается, но при этом еще не все треды завершили свою работу?

`join()` не завершает поток. Заджойненый тред ждет, пока отработает main

`detatch()` также не убивает поток. Фактически он сообщает `std::thread` что этот поток должен продолжать работать даже после уничтожения объекта `std::thread`. В деструкторе `std::thread` C++ проверяет, присоединён ли поток или отсоединён, и завершает работу программы, если проверка не пройдена.

Все треды должны быть заджойнены или задетачены (завершены) перед завершением main. Если условие не выполняется, то деструктор класса thread вызывает SIGABRT

33.

Что такое race condition?

> **Race condition (состояние гонки)** — это **ситуация, которая возникает в многозадачных или многопоточных программных системах**, когда результаты выполнения программы зависят от того, в какой последовательности выполняются инструкции или операции в различных потоках или процессах.

Состояние гонки возникает, когда несколько потоков или процессов одновременно пытаются получить доступ к одним и тем же ресурсам (например, переменным памяти, файлам или устройствам ввода-вывода) без синхронизации.

Пример состояния гонки: два потока пытаются изменить одну и ту же переменную. Если эти изменения не синхронизированы правильно, результаты могут быть непредсказуемыми, в зависимости от того, какой поток завершит операцию первым.

Состояние гонки может привести к непредсказуемому поведению программы или к ошибкам в её выполнении.

Приведите пример, когда возникает UB из-за одновременного изменения одних и тех же данных из разных тредов.

```
std::vector<int> v;

void job() {
  for (int i = 0; i < 100000; ++i) {
    v.push_back(i);
  }
}
```

```
int main() {
  std::thread t(job);
  job();
  t.join();
}
```

Падаем. Если не упали, то открываем пиво всей общагой

Что такое мьютекс? Приведите пример решения проблемы race condition с помощью мьютекса.

> mutex - объект, который позволяет получить эксклюзивный доступ к какому-то участку кода. Простейший примитив синхронизации.

```
std::vector<int> v;
std::mutex m;

void job() {
  m.lock();
  // Critical section
  for (int i = 0; i < 100000; ++i) {
    v.push_back(i);
  }
  m.unlock();
}

int main() {
  std::thread t(job);
  job();;
  t.join();
}
```

mutex может находиться в двух состояниях: в залоченном и в разлоченном. Пока mutex залочен, никто другой не сможет вызвать успешно lock() (треды, вызывающие lock() для залоченного mutex'a, блокируются)

Что такое deadlock и как он может возникнуть?

Интернет говорит, что

> **Deadlock** — это ситуация, при которой два или более потока (или процесса) блокируют друг друга, ожидая освобождения ресурса, который удерживается другим участником. В результате работа программ зацикливается, и они не могут продолжить выполнение.

Мещерин говорит, что

> Deadlock - ситуация, когда по какой-либо причине не был сделан unlock (например, вылетело исключение в критической секции). В таком случае решение - RAII над mutex. (std::lock_guard)

34.

Как реализовать std::thread, используя сисколлы?

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

```cpp
#include <sched.h>
#include <sys/wait.h>

class Thread {
  using Callable = void(*)();
public:
  Thread(Callable func): func(func) {
    stack = mmap(
        NULL, STACK_SIZE, PROT_READ|PROT_WRITE,
        MAP_PRIVATE|MAP_ANONYMOUS|MAP_STACK, -1, 0);
    pid = clone(
        threadRoutine, stack,
CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|
                        CLONE_SYSVSEM|CLONE_SETTLS|CLONE_PARENT_SETTID|
                        CLONE_CHILD_CLEARTID, (void*)func);
  }

  void join() {
    int status;
    waitpid(pid, &status, 0);
    pid = -1;
  }

  ~Thread() {
    if (pid > 0) {
      std::terminate();
    }
    munmap(stack, STACK_SIZE);
  }
private:

  static int threadRoutine(void* arg) {
    Callable routine = reinterpret_cast<Callable>(arg);
    routine();
    return 0;
  }

  static const int STACK_SIZE = 8392704;
  Callable func;
  void* stack;
  int pid = -1;
};

void f() {
  for (int i = 0; i < 100; ++i) {
    std::cout << i << " ";
  }
  std::cout << std::endl;
}

int main() {
  Thread t(f);
  f();
  t.join();
}
```

Передача аргументов в функцию осуществляется через type erasure
Падает с сегфолтом, гпт говорит, что

1. должен передаваться указатель на конец стека
2. waitpid не работает (При использовании `CLONE_THREAD` поток не является отдельным процессом, а частью текущего процесса, поэтому `waitpid` не применим), вместо него он использует futex

Как пользоваться сисколлом clone и какие у него есть параметры?

```
int clone(int (*fn)(void *), void *stack, int flags, void *arg, ...
    /* pid_t *parent_tid, void *tls, pid_t *child_tid */ );
```

- **fn** - что исполнять
- **stack** - указатель на стек для дочернего процесса. Обычно просто выделяется новый
- **flags** - набор флагов. Указывают что именно склонировать, определяют запустится новый процесс или тред. Выглядят как CLONE_...
- **arg** - указатель на то, что передается в вызываемую функцию в качестве аргументов

Покажите, как надо вызывать сисколл clone, чтобы создать полноценный std::thread

```
// также стек создает в виде одной страницы без каких-либо прав для защиты
от угрозы перезатирания важной информации злоумышленниками
stack = mmap(NULL, STACK_SIZE, PROT_READ|PROT_WRITE,
                MAP_PRIVATE|MAP_ANONYMOUS|MAP_STACK, -1, 0);
clone(func, stack, CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|

CLONE_THREAD|CLONE_SYSVSEM|CLONE_SETTLS|CLONE_PARENT_SETTID|
                        CLONE_CHILD_CLEARTID, /*arg*/);
// чтобы передать аргументы надо пострадать
```

Что из себя представляют треды с точки зрения ОС?

В Linux треды реализуются через **LWP (Lightweight Processes)**, которые представляют собой сущности ядра с почти теми же свойствами, что и процессы. Основное отличие — треды в одной группе делят ресурсы, тогда как процессы изолированы.

Что такое тред-группа?

> **Тред-группа** — это совокупность всех потоков, принадлежащих одному процессу в Linux. Все потоки, которые создаются процессом, образуют одну тред-группу. Главный процесс в этой группе называется **лидером тред-группы**

- Если любой из тредов в группе вызывает execve, то все треды в группе завершаются, кроме главного, и новая программа исполняется в главном треде
- Если один из тредов делает fork, то каждый тред в группе может делать wait на ребенка

Что такое tid, tgid, как их узнать, в чем разница с pid?

> **tgid** - общий идентификатор группы потоков, совпадает с PID основного процесса.
> **tid** - уникальный идентификатор для каждого потока.
> **pid** - идентификатор процесса. Общий для всех потоков.

Можно узнать в htop, если выставить в настройках отображение соответствующих колонок. Только htop может вместо pid показывать tid. В таком случае нужно нажать H. Но там нельзя увидеть TID, короче кринж, используем ельф!

```
ps -eLf # LWP отвечает за TID, TGID совпадает с PID
```

Как послать сигнал отдельному треду?

```
int tkill(int tid, int sig);
```

35.

Что такое ассемблер? Какие есть разновидности ассемблера?

> Ассемблер - язык команд процессора

Ассемблеры различаются в зависимости от архитектуры процессора и особенностей языка:

1. **По архитектуре:**
   - **x86/x86-64**: Используется для процессоров Intel и AMD. Распространённые ассемблеры — NASM, MASM, GAS.
   - **ARM**: Используется для мобильных устройств, встраиваемых систем и серверов. Ассемблер ARM имеет компактный и эффективный синтаксис.
   - и прочие
2. **По синтаксису:**
   - **Intel-синтаксис** (NASM, MASM): Более читаемый для новичков, например, в x86 используются ключевые слова `mov eax, ebx`.
   - **AT&T-синтаксис** (GAS): Используется в GNU Assembler, отличается префиксами `%` для регистров и `$` для констант (`movl %eax, %ebx`).

Что такое регистры? Перечислите основные регистры в архитектуре x86 и их предназначение

> Регистры - ячейки памяти в процессоре

1. rax - аккумулятор, используется для хранения результатов арифметических операций
2. rbx - base, используется для хранения адресов или данных.
3. rcx - counter, используется как счетчик в цикле.
4. rdx - data, для умножения, деления и операций с большим объемом данных.
   Со временем разрядность процессора увеличиливась: с 8 бит до 64. В связи с этим регистры менялись. Поэтому ax - имя регистра в 16-битном представлении. eax - в 32-битном, rax - в 64-битном. Причем все еще можно обращаться к ячейке разрядностью 16 или 32 бита, используя ax (младшие 8 бит) или eax

Расскажите про основные ассемблерные инструкции и их синтаксис: mov, арифметические инструкции, логические инструкции.

mov:

```
mov    to, from
mov    [to], from // разыменовать to и положить туда from
```

+:

```
int x = 4;
int y = 2;
```

```
x += y:

mov     eax, DWORD PTR [rbp-8]
add     DWORD PTR [rbp-4], eax
```

```
long long x = 4;
int y = 2;

mov     eax, DWORD PTR [rbp-12]
cdqe
add     QWORD PTR [rbp-8], rax
// положили значение x в eax - четырехбайтный регистр
// cdqe - "перемещаем из eax в rax". Фактически - умножение на 2
```

- перемещаем слагаемое в eax и добавляем к левому операнду
- аналогично sub (вычитание)

*****:

```
int x = 4;
int y = 2;
x *= y:

mov     eax, DWORD PTR [rbp-4]
imul    eax, DWORD PTR [rbp-8]
mov     DWORD PTR [rbp-4], eax
```

- умножать можно только то, что лежит в регистре, поэтому двух команд недостаточно

/:

```
int x = 4;
int y = 2;
x /= y;

mov     eax, DWORD PTR [rbp-4]
cdq
idiv    DWORD PTR [rbp-8] // делит число, лежащее в регистре a
mov     DWORD PTR [rbp-4], eax
```

- делить можно только то, что лежит в регистре
- целая часть кладется в eax, остаток - в edx

Умножение/деление на константы

mul и div - дорогие инструкции. Вместо них компилятор старается их заменить легковесными sub, add и операциями побитового сдвига

```
x *= 5

mov     edx, DWORD PTR [rbp-4]
mov     eax, edx
sal     eax, 2 // побитовый сдвиг влево на 2, т.е. умножение на 4
add     eax, edx
mov     DWORD PTR [rbp-4], eax
```

```
// деление весело преобразуется
x /= 3;

mov     eax, DWORD PTR [rbp-4]
movsx   rdx, eax // одновременно копирует и расширяет
imul    rdx, rdx, 1431655766
mov     rcx, rdx
shr     rcx, 32 // побитовый сдвиг вправо
cdq
mov     eax, ecx
sub     eax, edx
mov     DWORD PTR [rbp-4], eax
```

Логические инструкции

```
and    operand1, operand2
or     operand1, operand2
xor    operand1, operand2
not    operand
```

Результаты записываются в регистр флагов

36.

Инструкции безусловного и условного перехода в ассемблере. Регистр флагов.

```
jmp     .L2 // безусловный переход
```

```
if (x > 4) {
  goto mylabel;
}

cmp     DWORD PTR [rbp-4], 4
jle     .L4    // jump less equal, перепрыгиваем if
jle     .L2    // тело if'a
```

- результат сравнения записывается в регистр флагов, который состоит из 64 бит, и каждый бит отвечает за какое-либо событие
- условные переходы проверяют значения одного или нескольких флагов в регистре флагов (CF, OF, PF, SZ, ZF), которые устанавливаются в результате выполнения предыдущей команды (например cmp или test)

1. Беззнаковые переходы (unsigned)

Используются для работы с беззнаковыми числами (например, после операций с `cmp`).

Инструкция	Условие	Флаги	Описание
je/jz	Равно (equal)	ZF = 1	Переход, если равно
jne/jnz	Не равно (not equal)	ZF = 0	Переход, если не равно
jb/jnae	Меньше (below)	CF = 1	Переход, если меньше
jnb/jae	Не меньше (above/equal)	CF = 0	Переход, если не меньше

Инструкция	Условие	Флаги	Описание
ja/jnbe	Больше (above)	CF = 0 и ZF = 0	Переход, если больше
jbe/na	Не больше (below/equal)	CF = 1 или ZF = 1	Переход, если не больше

---

2. Знаковые переходы (signed)

Используются для работы со знаковыми числами (например, после операций с `cmp`).

Инструкция	Условие	Флаги	Описание
jl/jnge	Меньше (less)	SF ≠ OF	Переход, если меньше
jge/jnl	Не меньше (greater/equal)	SF = OF	Переход, если не меньше
jg/jnle	Больше (greater)	ZF = 0 и SF = OF	Переход, если больше
jle/jng	Не больше (less/equal)	ZF = 1 или SF ≠ OF	Переход, если не больше

---

3. Переходы для конкретных флагов

Используются для проверки отдельных флагов.

Инструкция	Условие	Флаги	Описание
jc	Переход при переносе	CF = 1	CF установлен
jnc	Переход без переноса	CF = 0	CF не установлен
jo	Переполнение	OF = 1	OF установлен
jno	Без переполнения	OF = 0	OF не установлен
js	Отрицательный результат	SF = 1	SF установлен
jns	Положительный результат	SF = 0	SF не установлен
jp/jpe	Чётный результат	PF = 1	PF установлен
jnp/jpo	Нечётный результат	PF = 0	PF не установлен

Как написать аналоги if, while и for на ассемблере?

if

- Если условие не выполняется, перепрыгиваем тело. Тело if'a расположено под проверкой условий

```
if (x < y) { x = y; }
int z = 0;


mov     eax, DWORD PTR [rbp-4]
```

```
cmp     eax, DWORD PTR [rbp-8]
jge     .L2
mov     eax, DWORD PTR [rbp-8]
mov     DWORD PTR [rbp-4], eax

.L2:
mov     DWORD PTR [rbp-12], 0
```

for

```
for (int i = 0; i < 100; ++i) { ++x; }

mov     DWORD PTR [rbp-8], 0 // кладем на стек i
jmp     .L2

.L3:  // тело for
add     DWORD PTR [rbp-4], 1
add     DWORD PTR [rbp-8], 1

.L2: // метка с проверкой условия
cmp     DWORD PTR [rbp-8], 99
jle     .L3
```

while

То же самое что и for, только без счетчика

```
while (x < 100) { ++x; }

jmp     .L2

.L3:
add     DWORD PTR [rbp-4], 1

.L2:
cmp     DWORD PTR [rbp-4], 99
jle     .L3
```

37.

Как использовать в программе на Си ассемблерную функцию из другого файла? Покажите на примере функции проверки числа на простоту

```
section .text
global is_prime

is_prime:

  cmp  eax, 2
  jb   .not_prime  // хз что такое jb, может можно заменить на jl
  je   .is_prime

  test eax, 1
  jz   .not_prime

  mov  ecx, 3  // ecx -  счетчик в цикле
  mov  ebx, eax
```

```
.check_loop:
  mov  eax, ecx
  mul  eax    // если у mul один аргумент, результат кладется в eax
  cmp  eax, ebx
  jg   .is_prime

  mov  edx, 0
  mov  eax, ebx
  div  ecx
  cmp  edx, 0
  je   .not_prime

  add  ecx, 2
  jmp  .check_loop

.is_prime :
  mov  eax, 1
  ret

.not_prime:
  xor  eax, eax
  ret
```

```
#include <iostream>
// не скомпилируется без extern "C", возможно проблема в манглировании имен
extern "C" int is_prime(int n);

int main() {
  int number;

  std::cout << "Enter a number: ";
  std::cin >> number;

  int result = is_prime(number);
  if (result) {
    std::cout << number << " is a prime number" << std::endl;
  } else {
    std::cout << " is not a prime number" << std::endl;
  }
}
```

```
nasm -f elf64 is_prime.asm
g++ is_prime.cpp is_prime.o
```

Как с помощью gdb делать отладку ассемблерного кода? Как просматривать текущие значения регистров, как делать пошаговое исполнение ассемблерных инструкций?

```
g++ -g is_prime.cpp is_prime.o
gdb ./a.out
b is_prime
disassemble # выведет ассемблерный код, который предстоит выполнять процессору
b *0x0000555555555313 # поставить брейкпоинт на инструкцию
```

```
stepi # или si, шагает по ассемблерным инструкциям
info registers # значения регистров в текущий момент
```

38.

Инструкции call и ret

> **call** - работает аналогично jump'у за исключением того, что записывает на стек адрес возврата из функции - указатель на следующую инструкцию *(rip)*
> **ret** - инструкция, обратная call. Снимает со стека адрес возврата, интерпретирует считанные байты как адрес следующей инструкции и дальше исполняет код.

Что такое стековый фрейм?

Стек делится на фреймы, и стековый фрейм - кусок, который соответствует локальному скопу одной функции. Каждый фрейм имеет "базу" - хранится в rbp

Что такое stack pointer и base pointer, регистры ebp и esp?

> **rsp** - регистр, хранящий stack pointer, показывает на верхушку стека (конец на текущий момент)
> **rbp** - регистр, хранящий base pointer, который указывает на начало базы текущего фрейма

Что происходит на уровне ассемблера при вызове функций и при возврате из них?

Вызов функции:

1. Вызывается call, который кладет на стек **адрес возврата**
2. При заходе в функцию сохраняется **предыдущий rbp** на стеке
3. в rbp записывается значение из rsp
   Возврат из функции:
4. rbp снимается со стека и записывается в регистр rbp
5. вызывается ret

Где хранятся аргументы функций при вызове? Где хранится результат функции сразу после вызова?

1. rdi - 1-ый
2. rsi - 2-й
3. rdx - 3-ый
4. rcx - 4-ый
5. r8 - **5**-ый
6. r9 - **6**-ый
   Если аргументов у функции больше, то они положаться на стек перед адресом возврата.
   Результат также по возможности кладется в регистры, но если он слишком большой, то на стек.
   ==**Вопрос в зал:** почему раньше в примере is_prime использовался rax для передачи 1-го аргумента?.. ==
   Ну если в ассемблерном коде использовать 32-битные регистры, то первый аргумент передается в eax....

Что делает флаг компиляции -fno-omit-frame-pointer, зачем он нужен?

```
Сохранение всех начал фреймов (rbp) делает выполнимым backtrace
```

Но если backtrace не нужен, можно использовать оптимизацию, позволяющую не хранить base pointer `-fomit-frame-pointer`. И наоборот с флагом `-fno-omit-frame-pointer` компилятор сохраняет использование регистра `rbp` как frame pointer.

Что такое атака переполнения буфера и какие средства защиты от нее существуют? Что такое stack protector, для чего он используется, что делает флаг компиляции -fno-stack-protector?

> Когда в коде осуществляется набор символов в char*, утилиты для ввода (scanf, cin) не проверяют, сколько символов было введено. Поэтому в переменную можно ввести больше символов, и адрес возврата перезатрется
> Причем если адрес возврата будет валидным, указывающим в какое-то место, то ret пойдет тупо исполнять инструкции по новому адресу

Способы защиты:

- fstack-protector - флаг, заставляющий компилятор сгенерировать код, который проверяет не перезатерт ли адрес возврата. По сути включает канарейки.
  Принцип работы: компилятор кладет на стек после rbp число, которое известно только ему. Перед выходом из функции смотрит не поменялось ли оно.
- NX-бит: запрещает выполнение кода в сегментах памяти, предназначенных только для данных (например, стек и куча). Защищает в ситуации, если в буфер пишем ассемблерную инструкцию, а адрес возврата ставим на начало буфера
  fno-stack-protector логично что делает.

Как получить ошибку “Stack smashing detected”?

```cpp
#include <cstdio>

void f() {
  char buf[10];
  scanf("%s", buf);
}

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

```
./a.out и вводим что-то длинное
```

stack-protector - защита по умолчанию

40.

Кэши процессора. Сколько уровней кэша есть в процессоре, зачем они нужны?

> Кэш процессора (CPU cache) — это высокоскоростная память, встроенная в процессор, предназначенная для временного хранения данных и инструкций, которые часто используются. Он служит для уменьшения времени доступа к оперативной

> памяти (RAM), которая относительно медленная по сравнению с процессором. Кэш хранит копии данных из основной памяти, к которым процессор обращается наиболее часто или которые могут понадобиться в ближайшее время.

Уровни кэшей процессора:

- L1 - наименьший и самый быстрый тип кэшей. Находится прямо в процессоре. Одна часть хранит инструкции, вторая - данные. Размер - от 2Кб до 64Кб (в современных процах для каждого ядра)
- L2 - чуть больше и чуть медленнее L1. Может быть в процессоре, может быть снаружи. Может быть свой для каждого ядра, может быть общий (зависит от архитектуры). Размер обычно от 256Кб до 512Кб (для каждого ядра)
- L3 - самый большой и самый медленный уровень. Лежит снаружи процессора и доступен всем ядрам. Играет важную роль в общении ядер и передаче данных. Размер от 1Мб до 8Мб и выше

Как узнать размеры кэшей своего процессора?

```
lscpu
```

- **L1d cache:** Кэш данных уровня 1
- **L1i cache:** Кэш инструкций уровня 1
- **L2 cache:** Кэш уровня 2.
- **L3 cache:** Кэш уровня 3.
  У меня выводит суммарный кеш, можно посмотреть для одного ядра командой

```
cat /sys/devices/system/cpu/cpu0/cache/index0/size # L1d для ядра 0
cat /sys/devices/system/cpu/cpu0/cache/index1/size # L1i для ядра 0
cat /sys/devices/system/cpu/cpu0/cache/index2/size # L2 для ядра 0
cat /sys/devices/system/cpu/cpu0/cache/index3/size # L3 общий
```

Что такое кэш-линия? Почему делать обход матрицы по строкам эффективнее, чем по столбцам?

> **Кэш-линия (cache line)** — это минимальная единица данных, которая может быть загружена или сохранена в кэш процессора из основной памяти (RAM). В современных процессорах от 32 до 128 байт (часто 64).

При обращении к какому-то адресу, он либо лежит в каком-то из кешей и загружается во все предыдущие, либо, если не нашли в кеше, то подгружается соответствующая кеш-линия. Поэтому обходить матрицу по столбцам быстрее - в кеш-линию выгружается непрерывный кусок памяти, а не происходят прыжки по строкам в разные кускипамяти. Из этого следует то, что взять один инт из кэша - все равно что взять 16 подряд лежащих

41.

Что такое branch prediction, в чем его идея?

Процессор пытается наперед подгрузить инструкции, но при if неизвестно, что нужно подгружать. Вычисления происходят гораздо медленее, чем загрузка следующих интсрукций, так что процессор загружает ветку, которая кажется ему наиболее вероятной, наперед и обрабатывает их (раскодирует, преобразует и тд)

Если сценарии примерно равновероятны, то процессор ошибается в половине случаев и ему приходится менять уже загруженные инструкции. Если одна ветка выполняется почти всегда, то процессор запоминает ее как более вероятную.

Как подсказать компилятору (средствами C++20, а также без него), какая из веток if более вероятна? Как это отразится на ассемблерном коде?

```
// аттрибуты c++20
if (...) [[likely]] {}

if (...) [[unlikely]] {}

// до c++20, 1 означает, что ветка более вероятна
if (__builtin_expect(x > 0, 1)) {}
```

Компилятор может использовать эту информацию, чтобы:

1. Организовать более вероятную ветку так, чтобы она была ближе к основному потоку выполнения.
2. Уменьшить число инструкций перехода для вероятной ветки.

42.

Как делать ассемблерные вставки в коде на Си? Расскажите о синтаксисе в общих чертах

Ассемблерные вставки делаются через команду **asm()** . Обычно вместе со вставками используют ключевое слово volatile , чтобы компилятор не смог ничего соптимизировать.

Аргументы функции asm():

- Команда - сишная строка, внутри нее через % кладутся номера аргументов - переменных, которые передадутся в функцию дальше, а через $ - просто численные константы
- Далее через двоеточие переменные, куда записывать результат
- Затем через двоеточие переменные, откуда брать данные

Как с помощью ассемблерной вставки узнать количество тактов процессора, прошедшее между двумя данными строчками кода?

Регистр **rdtsc** сохраняет информацию о количестве тактов с момента последнего сброса процессора. При помощи ассемблерной вставки можно вычислить количество тактов, прошедших во время выполнения программы
Плюсы такого подхода: вызовы библиотечных функций сами по себе долгие, так что такой способ дает точный результат

43.

Что такое кольца защиты?

Кольца защиты (Ring Protection Levels) — это механизм управления доступом к системным ресурсам в архитектуре процессора, который используется для разграничения уровней привилегий между различными частями операционной системы и приложениями. Кольца защиты предоставляют разные уровни прав доступа к ресурсам системы, таким

как память, устройства и выполнение кода, для предотвращения нежелательного или вредоносного воздействия.

- **Ring 0 (Кольцо 0) — Привилегированный режим (kernel mode):**
  - На этом уровне работает ядро операционной системы (kernel), которое имеет полный доступ ко всем системным ресурсам, включая оборудование.
- **Ring 1 (Кольцо 1) — Полупривилегированный режим:**
  - Это промежуточный уровень, который используется для драйверов и других системных приложений, которым необходим доступ к некоторым системным ресурсам, но не полный доступ как в кольце 0.
- **Ring 2 (Кольцо 2) — Полупривилегированный режим:**
  - Это также промежуточный уровень, но с ещё меньшими правами, чем у кольца 1.
  - Он редко используется в современных системах, но может быть задействован для специализированных задач и драйверов, которые требуют более ограниченного доступа.
- **Ring 3 (Кольцо 3) — Пользовательский режим (user mode):**
  - Это наименее привилегированный уровень.
  - На этом уровне выполняются обычные пользовательские приложения, которые имеют доступ только к ограниченному набору системных функций.
  - Процесс в кольце 3 не может напрямую обращаться к аппаратным ресурсам или выполнять привилегированные инструкции.

Какие есть режимы работы у процессора и чем они отличаются? Что такое привилегированные инструкции процессора и что к ним относится?

Режимы работы процессора:

- Привилегированный
  В этом режиме доступно больше инструкций, например:
  1. HLT - приостановить процессор
  2. INVLPG - инвалидировать запись в TLB кэше
  3. LIDT - загрузить таблицу прерываний. По ней процессор узнает, куда прыгать в случае прерывания с тем или иным сигналом
  4. MOV CR - операции надо контрольными регистрами - регистрами, доступными только в этом режиме. CR0 - хранит флаг, используется ли сейчас виртуализация памяти. CR2 - хранит адрес куда пойти при page fault.
     Также в этом режиме отключается виртуализация памяти - система ходит напрямую в адреса в памяти
- Обычный
  Тут доступны не все инструкции
  В привилегированном режиме исполняется только операционная система. Все пользователи работают в обычном режие

Чем принципиально отличается вызов сисколла от вызова обычной функции? Что делает ассемблерная инструкция syscall?

Syscall - особая инструкция, это настоящий сисколл. Все что использовали до этого - обертки. Не используем напрямую syscall, т.к. для нее предварительно нужно положить аргументы и еще пару вещей в правильные регистры.

Отличается она тем, что при ее исполнении процессор переключается в привилегированный режим, открывает новый стек и идет по определенному адресу выполнять сисколл. При выходе возвращает все как было.

Можно увидеть инструкцию вообще везде, если слинковать статически

```
g++ -static syscall.asm
objump -d a.out > output.txt
```