«Низкоуровневое программирование»
МФТИ

Андреев Кирилл при поддержке Колесникова Дениса

Преподаватель: Илья Мещерин

Осень 2024

Содержание

Раздел 1. Линковка и библиотеки. Понятие сисколлов

Пункт 1

Сборка проекта на `c++` включает в себя 4 стадии:

1. Препроцессинг – раскрытие директив препроцессора (`include, define...`). Для выполнения всех стадий до него флаги: `g++ -E test.cpp > test_preprocessed.cpp`
2. Компиляция – преобразование кода на `c++` в ассемблерный. `g++ -S test.cpp > test.s`
   Выполняется программой (вместе с препроцессингом)
   `/usr/libexec/gcc/x86_64-linux-gnu/13/cc1plus`
3. Ассемблирование – преобразование ассемблерного кода в машинный. Получаем объектный файл. `g++ -c test.cpp or test.s > test.o`
   Выполняется программой `as`
4. Линковка – связывание нескольких объектных файлов. Находятся определения функций из внешних библиотек.
   Выполняется программой `ld` (на самом деле `/usr/libexec/gcc/x86_64-linux-gnu/13/collect2`)

Чтобы увидеть все этапы сборки нужно использовать флаг – `g++ -v test.cpp`

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.

Чтобы дизассемблировать исполняемый файл можно воспользоваться утилитой: `objdump -d a.out`

Пункт 2

Линковка – связывание нескольких объектных файлов. Находятся определения функций из внешних библиотек. Она, в свою очередь, включает в себя 2 этапа – статический и динамический. Подробнее: Пункт 3.

Выполняется программой `ld` (на самом деле `/usr/libexec/gcc/x86_64-linux-gnu/13/collect2`)

Библиотеки (либы) - уже скомпилированные ELF файлы, в которой находятся символы, но уже с определениями. Общепринято называть либы начиная с **lib**. Либы бывают статические и динамические (подробнее - опять Пункт 3). Статические имеют расширение `.lib` или `.a`, динамические – `.so` или `.dll`.

Посмотреть свои либы можно по пути `/usr/include/c++/13/` (Хедеры) и `/lib/x86_64-linux-gnu/` (so-шники)

Пример линковки с помощью `ld`:

```cpp
// test.cpp
#include <iostream> // vim /usr/include/c++/11/iostream

int main() {
    int x;
    std::cin >> x;
    std::cout << x + 5 << '\n';
}
```

1. Сначала скомпилирем как обычно и с помощью `ldd` посмотрим зависимости от либ
2. Компилируем без линковки – `g++ -c test.cpp > test.o`
3. Линкуем со стандартными либами:
   `ld -o test test.o -lc /lib/x86_64-linux-gnu/libstdc++.so.6 \`
   `/lib/x86_64-linux-gnu/libm.so.6 /lib/x86_64-linux-gnu/libgcc_s.so.1` (Просто -lstdc++ не работает =(. Вероятно потому что оно не умеет искать с расширениями с версиями aka *.so.6)
4. С помощью `readelf -a a.out` можем убедиться, что это исполняемый ELF файл.
5. Сейчас он не запускается, так как запуск начинается с функции _start а мы собрали проект без "обёртки над мейном"+ у файла нет динамического линковщика вшитого. Чтобы это пофиксить выполняем пункт 2 с некоторой модификацией:
   `ld -o test test.o -lc /lib/x86_64-linux-gnu/libstdc++.so.6 -dynamic-linker \`
   `/lib64/ld-linux-x86-64.so.2 /lib/x86_64-linux-gnu/libm.so.6 \`
   `/lib/x86_64-linux-gnu/libgcc_s.so.1 -e main \` Здесь мы говорим что точка входа в программу – мейн `-e main` и добавляем стандартный динамический линковщик (путь может отличаться в зависимости от ОС)
   Но мы всё равно падаем с *Segmentation Fault* так как некорректно создаются переменные из других либ и в конце main есть `return` непонятно куда (функции-обёртки `_start` нету). Если в конце `main` поставить функцию `exit(0)`, то сегфолта не будет (но это не точно =/).

Пункт 3

Статическая линковка - Весь необходимый код из библиотек объединяется с объектным кодом программы в один исполняемый файл на этапе компиляции. - Это означает, что все функции из библиотек присутствуют в итоговом исполняемом файле, и программа не зависит от наличия библиотек на целевой системе.

Динамическая линковка - Код библиотеки не включается в исполняемый файл, а используется во время выполнения программы. - Это позволяет нескольким программам использовать одну и ту же библиотеку, что экономит дисковое пространство.

Применим утилиту для получения информации из ELF файла `readelf -a a.out` Найдём там 2 раздела – `.symtab` и `.dynsym` - таблицы символов.

`.symtab` – статическая линковка, до момента запуска программы, адреса проставляются на последней стадии сборки.

`.dynsym` – динамическая линковка, после запуска программы, это значит, что код вызываемой функции на самом деле написан где-то извне и нужно его подгрузить по мере надобности.

Разница между статической и динамической библиотеками заключается в том, что для статических библиотек компоновка выполняется во время компиляции, встраивая исполняемый код в ваш двоичный файл, в то время как для динамических библиотек компоновка выполняется динамически при запуске программы.

Чтобы принудительно слинковать файл статически, используем флаг **-static** при компиляции: `g++ -static test.cpp`. Размер файла значительно увеличится и у него не будет `.dynsym`. Это может быть полезно для запуска программы на других устройствах, где могут не быть установлены нужные библиотеки.

Чтобы посмотреть зависимость от динамических библиотек для данного исполняемого файла используем `ldd a.out`

Немного отсебятины про собрать с либой по нестандартному адресу – у меня оно статически собирает файл, если я стандартную либу куда-то перетянул (кидал прямо в ~/ - ему пофиг, он находит)

Чтобы собрать свою динамическую библиотеку, используем следубщий флаг `g++ -shared libawesome.cpp -o libawesome.so`

Допустим у нас есть по пути `/awesome-path/` либа `libawesome.so`. Тогда если файл `test.cpp` использует её, то проект можно собрать командой `g++ test.cpp -lawesome -L/awesome-path/`

Здесь -l<название библиотеки> и -L<путь где искать эту библиотеку>.

Но запустить прогу не получится, так как динамический линковщик не знает о таком рамположении либы. Для этого нужно прокинуть флаг, где дополнительно искать либу для линковщика. Делается это так: `g++ test.cpp -lawesome -L. -Wl,-rpath,.` С помощью флага -Wl прокидываем следующие команды линковщику. `-rpath ПУТЬ` – Задать путь поиска общих библиотек времени выполнения

`LD_LIBRARY_PATH` - это предопределенная переменная среды в Linux / Unix, которая задает путь, к которому компоновщик должен обращаться при связывании динамических библиотек / разделяемых библиотек. (По-русски - мувнули стандартную библиотке, теперь прога a.out не работает. Запускаем её с изменённой переменной `LD_LIBRARY_PATH`, куда указываем где нужно искать либу и всё работает)

Ex. `LD_LIBRARY_PATH=/lib/x86_64-linux-gnu/ha ./a.out`

Чтобы посмотреть, какие вызовы библиотечных функций делает данная программа в ходе выполнения, воспользуемся утилитой `ltrace`: `ltrace ls ~/` или можно подключиться к работающему процессу процессу `ltrace -p 27766`

Пункт 4

ELF – executable linkable format, формат бинарных файлов для linux систем (`.o` и `.out` в том числе)

Тип **ELF**-файла можно посмотреть с помощью команды `readelf -h file.o`

Три типа `ELF`-файлов:

- **relocatable** – пример неслинкованный объектный файл. `g++ -c test.cpp`. В хедере увидим тип `REL (Перемещаемый файл)`
- **executable** – пример обычный `a.out`. В хедере увидим `DYN (Position-Independent Executable file)`
- **shared** – пример динамическая библиотека (формат .so). В хедере увидим `DYN (Совм. исп. объектный файл)`

Каждый ELF файл разбивается на секции. Основные секции:

1. В начале *ELF Header* (Заголовок бинарного файла, в котором расположена основная информация о файле)
   В нем находятся в том числе:
   (a) Magic – специальный байты, с которых начинается файл (договоренность, что именно с таких-то специальных байт должен начинатся ELF-файл, для расшифровки его утилитами; например, для файлов `.jpeg` начало будет уже другим)
   (b) Class – формат файла (ELF64)
   (c) Data – способ кодиравания в байты
   (d) OS/ABI – для какой системы файл
   (e) Type – тип файла (DYN, Position-Independent Executable file)
   (f) И другая техническая информация
2. Секция `.text` – бинарный код программы, загружаемый в оперативную память на исполнение
3. Секция `.data` – проинициализированные статическая переменные программы
4. Секция `.rodata` – read-only data, то же самое что и (3) только нельзя изменять (например литерал строки в коде)

5. Секция `.bss` – статические переменные, которые будут проинициализированы в runtime
6. Секция `.symtab` – таблица символов, которая разделяется в свою очередь на `.dynsym` и `.symtab`

Утилиты для чтения `ELF`-файлов:

- `readelf` – показать информацию о данном файле
- `hexdump` – бинарное (побайтовое) прочтение `.o` файла
  `-c` – для расшифровки соотвествующих байт в символы
- objdump – тоже показывает информацию о файле
  `-t` – вывод таблицы символов
  `-d` – дизассемблирование

Утилита `objcopy` используется для копирования и преобразования объектных файлов. Она позволяет изменять формат, изменять содержимое, а также экспортировать и импортировать определенные секции из объектных файлов и исполняемых программ. Основные функции `objcopy`:

- Конвертация форматов файлов (например, из ELF в бинарный и обратно).
- Удаление ненужных секций или символов.
- Создание статических библиотек из объектных файлов.
- Извлечение определенных секций, например, только исполняемого кода.

Ex. `objcopy --strip-debug libawesome.so libawesome_stripped.so`

Удалит все отладочные символы, делая библиотеку легковесной. Первый параметр – название input файла. Второй – output (иначе сохранит туда же).

Пункт 5

Символы – все те сущности, которыми оперирует линковщик - переменные (OBJECT), в том числе функции (FUNC), о существовании которых знает линковщик (неформальное определение).

Манглирование – процесс преобразования имени объекта из программы, написанной на C++ в имя, с которым далее будет работать линковщик (в обратную сторону - **деманглирование**). Следует следующим правилам (на примере *_ZSt3cin*):

- С *_Z* начинается каждый символ, определенный пользователем.
- *St* – означает *std::*

- *3* – количество символов в названии
- *cin* – само название символа

Чтобы по манглированному имени восстановить исходное, можно воспользоваться утилитой `c++filt`. Можно поставить флаг `-t` для получения имени типа (`c++filt -t i` получим `int`)

Чтобы посмотреть список символов в данном `ELF`-файле, воспользуемся командой `readelf -s a.out`

В таблице символов, можно увидеть следущие типы символов (`BIND` столбец): `LOCAL`, `GLOBAL` и `WEAK`.

`WEAK` символы – ведут себя также как и `GLOBAL` символы, однако если встретится символ с таким же именем, но без пометки `WEAK` (`GLOBAL` символ), то он перезатирается этим другим символом, что происходит на стадии линковки. Соответственно `GLOBAL` символ - *strong*.

Пример `WEAK` символа – оператор `new`.

`static` примененное к глобально определенным функциям или переменным значит сделать этот объект локально видимым, то есть невидимой извне, или же приватной для этого файла. И наоборот, не примененное ключевое слово `static` оставляет символ глобальным. Из этих рассуждений возникают соответствующие символы – `LOCAL` и `GLOBAL`.

Ключевое слово `inline` вообще скрывает от линковщика сущность (в том числе помогает избавиться от конфликта имен).

Видимость символов в C++ может быть внешней и внутренней. Очень подробно статья на Хабре. Чтобы сделать символ приватным для внешних файлов (внутренне связаным) нужно объявить его `static` или положить в анонимный `namespace`.

Релокации в контексте C++ — это процесс назначения адресов для зависимого от позиции кода и данных программы и корректировки кода и данных для отражения назначенных адресов.

Когда компилятор генерирует машинный код, он в каждое место, в котором происходит либо вызов функции, либо загрузка адреса переменной, вставляет несколько байт, достаточных для записи в это место реального адреса переменной или функции, а также генерирует релокацию. Он не может сразу записать реальный адрес, поскольку на этом этапе ему этот адрес неизвестен. Функции и переменные после линковки могут оказаться в разных секциях, в разных местах секций, в конце концов секции могут быть загружены по разным адресам во время выполнения.

Релокация содержит информацию:

- По какому адресу нужно записать адрес функции или переменной
- Адрес какой функции или переменной нужно записать
- Формулу, по которой этот адрес должен быть посчитан
- Сколько байт зарезервировано под этот адрес

Пункт 6

Что происходит до и после `main`? Вообще, можно посмотреть ассемблер (команда `objdump -d a.out`). Глобально – над `main` есть функция-обёртка *_start*. В ней инициализируются глобальные константы, переменные (в том числе вызывается функция `ios_base::init` для `cin/cout`). В целом – инициализация библиотек, память под них и т.п. После `main` уничтожаются глобальные объекты. В самом конце вызывается функция `exit(int)`, которая корректно завершает процесс.

UPD: Ещё до вызова `main, _start`, создания переменных, динамический линковщик загружает в оперативку файлы библиотек и и маппит их в адреса нашей программы. (с помощью `strace ./a.out` можно в этом убедиться).

Чтобы не генерировать функцию `_start`, а начинать программу сразу с `main` нужно воспользоваться флагом компиляции `g++ -e main hello.cpp` (на самом деле функцию `_start` оно сгенерирует, но начнёт программу действительно с `main`).

UPD: Можно добавить флаг `-nostartfiles`, тогда `_start` действительно не будет.

Если запустить такую прогу, то она упадёт с сегфолтом (подробнее Пункт 2)

Чтобы это исправить, в конце `main` нужно вызвать функцию `exit(0)` или сисколл `_exit(0)`. (У меня оно всё равно падает, наверно из-за версии g++, если есть `cin/cout`. Они оказываются непроинициализированны. Но если их убрать и поставить сисколлы `read/write`, то всё ок, работает корректно).

UPD: решено в Пункте 2 правильной линковкой (добавлением ещё либ, то есть всех, которые показывает `ldd`).

Пункт 7

Дебаг - посмотреть за исполнением программы (построчно, по каким-то моментам и много чего ещё).

Сборка в debug режиме - используем флаг `g++ -g test.cpp`. Отличие от обычной в том, что такой исполняемый файл содержит отладочные символы - информация о том, какая строчка из исходного кода чему соответствует и другую информацию. Такой бинарь будет весить в несколько раз больше обычного.

Отлаживать программу можно с помощью `gdb`. Хороший файлик от Мещерина со всеми необходимыми командами тут.

Некоторые команды:

- Выбрать файл для отладки – `file <filename>`
- Начать отладку – `run`
- Поставить breakpoint на строчку кода – `b <номер строчки>` или `break <номер строчки>`

- Поставить breakpoint на функцию - так же как и на строчку, только указываем имя функции.
- Шаг без захода в функцию – `n`
- Шаг с заходом в функцию – `s`
- Вывести текущее значение переменной – `p <название переменной>` или `print <название переменной>`
- Посмотреть backtrace от текущего места исполнения – `bt`

Фраза “core dumped” означает, что в момент падения программы ОС поместила образ ядра (состояние) в файлик для информации о том, что происходило в момент подения и до него. Такие файлы у меня располагаются по пути `/var/lib/systemd/coredump/`. У кого так же - почитайте как с этим работать тут. У себя можете найти, прочитав ответ на StackExchange. По умолчанию он сохраняет файлы размером до 0 байт. Чтобы увеличить размер, воспользуемся командой `ulimit <количество байт>` или `ulimit -c unlimited` для безлимитного хранения.

Чтобы посмотреть с помощью gdb содержимое coredump-файла после того, как программа упала воспользуемся командой `gdb ./a.out /var/lib/systemd/coredump/coredump`

Пункт 8

Системные вызовы – функции, осуществляющие взаимодействие между приложением и операционной системой (Интерфейс, предоставляемый операционной системой, для работы с ней). **POSIX** (Portable Operating System Interface) – стандартизация системных вызовов, предоставляемых операционной системой.

`strace` – посмотреть в режиме реального времени те системные вызовы, которыми оперирует передаваемая на вход программа во время своей работы (`-p` - указать `PID` процесса, системные вызовы которого хотим посмотреть).

Документация, предоставляемая командой `man` подразделяется на разделы, например:

1. 1 – стандартные функции (за которыми мы и лезли в `man` обычно)
2. 2 – нужные нам для изучения системные вызовы, предоствляемые ядром операционной системы
3. 3 – библиотечные функции

Для того, чтобы брать из документации сущность из конкретного раздела (например сисколлов), можно указать этот раздел: `man 2 write`.

```
1 #include <unistd.h>
2 #include <cstdlib>
```

```cpp
#include <errno.h>
#include <string>
#include <cstring>
int main() {
    char buffer[100] = {'H', 'e', 'l', 'l', 'o', ',', ' '};
    auto taken = read(0, buffer + 7, 20);
    if (taken == -1) {
        auto errno_str = std::to_string(errno);
        write(2, errno_str.data(), strlen(errno_str.data()));
        exit(1);
    }
    taken += 7;
    for (int i = taken; i > 0;) {
        auto wrote = write(1, buffer + taken - i, i);
        if (wrote == -1) {
            auto errno_str = std::to_string(errno);
            write(2, errno_str.data(), strlen(errno_str.data()));
            exit(1);
        }
        i -= wrote;
    }
}
```

Тут пример использования `read/write`. Возвращаемые значения используем для индикации ошибки и конца ввода. Если возвращают -1, значит есть ошибка. Смотрим её код в глобальной переменной `errno`. По-хорошему `read` тоже оборачиваем в цикл и проверяем дополнительно, что вернулось 0 – это означает конец ввода.

`errno -l` – список с пояснениями всех возможных значений `errno`.

Раздел 2. Файлы и файловые системы

Пункт 9

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

По умолчанию для каждого процесса ОС выделяет 3 дескриптора:

- 0 – стандартный поток ВВОДА
- 1 – стандартный поток ВЫВОДА
- 2 – стандартный поток ВЫВОДА ОШИБОК

`int open(const char *pathname, int flags, .../* mode_t mode */)` – открывает файл, возвращает наименьший незанятый файловый дескриптор или -1 в случае ошибки.

Последний параметр отвечает за разрешённые действия над файлом при его создании (пишем 0666 - разрешить чтение и запись для всех). Подробнее о правах доступа в другом пункте. Пример вызова: `int fd = open("./input.txt", O_RDONLY | O_CREATE);` – открываем файл только на чтение; если такого файла нет, то создаём его.

`close` – закрывает файловый дескриптор (аналогия с `new/delete` для памяти). Возвращает 0 в случае успеха и -1 в случае неудачи. Принимает один параметр - номер дескриптора.

`lseek(fd, offset, whence)` – чтение с определенного места файла, где вторым аргументом задается сдвиг, а последним его характер:

1. `SEEK_SET` – чтение с номера байта
2. `SEEK_CUR` – сдвиг относительно текущей позиции
3. `SEEK_END` – сдвиг относительно конца файла

Возвращает в случае успеха тукущий offset в байтах с начала файла, в противном случае -1.

Если пытаться записать на сдвиг, выходящий за границы файлов, то между концом файла и записываемыми данными образуется "дырка". Если перекопировать содержимое этого файла в другой, то фиктивные байты заместятся уже настоящими нулевыми ('\0'), занимающими теперь память.

Реализация ср сами, там не сложно)))

Пункт 10

Чтобы перенаправить вывод команды в файл с перезаписью используем `>` – `echo 'Hello' > log.txt`

Так же можно использовать pipe | – он перенаправялет stdout левой команды в stdin правой. `./a.out | grep 'error'`

Без перезаписи, в режиме добавления >> – `echo 'My friend!' >> log.txt`

Чтобы перенаправить вывод одного из потоков (`cout` или `cerr`) в файл или в другой поток, воспользуемся тем же `t` но явно укажем файловый дескриптор потока, который хотим перенаправить (1 для `cout`, 2 для `cerr`): `./a.out 1> stdout.txt 2> stderr.txt`

Чтобы подавить вывод какого-то из потоков, перенаправим его в `/dev/null` (бездна) – `./a.out 2> /dev/null`

`tee` – команда, позволяющая продублировать вывод файла в стандатный поток в еще один источник – `./a.out | tee result.txt` – Мы увидим вывод и в консоли и в файле.

`int dup(int oldfd)` – системный вызов, создающий новый файловый дескриптор, означающий то же самое (наименьший доступный номер). При изменении `offset` при записи по старому дескриптору, он поменяется и по новому.

`int dup2(int oldfd, int newfd)` – то же самое, что и `dup`, но мы сами выбираем номер нового дескриптора – `newfd`. Если `newfd` уже открыт, то `dup2` его сначала закрывает (тихо, если возникают ошибки при закрытии, то `dup2` не сообщит о них =).)

Пункт 11

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 вам дерево файлов. Посмотреть какая файловая система стоит у вас можно с помощью команды `df -T -h` – флаг `-h` для красивого формата памяти (не в байтах). Во втором столбце будет написана ФС.

Наиболее популярные файловые системы

- Windows – `FAT, NTFS, exFAT`
- macOS – `HFS, APFS, HFS+`
- Linux – `EXT2, EXT3, EXT4, XFS, JFS`

Различают 7 основных видов файлов (можно посмотреть через `ls -l` рядом с правами):

1. `-` – обычные файлы
2. `l` – символические ссылки (можно создать командой `ln -s test.txt link.txt`)
3. `d` – директории

4. `p` – `FIFO` файлы (можно создать командой `mkfifo fifo_example`)
5. `s` – `Socket` файлы (можно создать командой `nc -lU socket.sock`) Затем Ctrl + C (оно сразу начинает слушать).
6. `b` – файлы, соотвествующие блочным устройствам (для общения с ними через `write` и `read`)
7. `c` – файлы, соотвествующие символьным устройствам
   Оба типа можно посмотреть по пути `/dev/`

Директории с точки зрения файловой системы – это особый вид файлов. Они не занимают место на жёстком диске. Так же как и обычные файлы, имеют права на чтение `r` и исполнение `x`. Но в случае с директориями правило на чтение означает возможность посмотреть её содержимое (например командой `ls`), но не зайти в неё. Право на исполнение наоборот, позволяет зайти в директорию (командой `cd`), но не увидеть названия всех файлов. НО если мы знаем точное название файла и у директории есть право на исполнение, то мы можем взаимодействовать с этим файлом.

На самом деле имена всех файлов это не идентификатор на жёстком диске. Это "жёсткие ссылки"(просто имена. Может быть несколько имён на одну и ту же область на жёстком диске). Настоящий идентификатор файла это `inode` – уникальное число для сегмента памяти на диске (для вашего файла). Чтобы посмотреть все `inode` для данного файла, воспользуемся командой `ls -i`

Чтобы создать ещё одну жёсткую ссылку на файл, используем команду `ln test.txt hardlink.txt`

Под капотом оно использует сисколл `link`

Соответственно, чтобы удалить жёсткую ссылку, используется сисколл `unlink`

При этом файл действительно удаляется с диска, когда количество жёстких ссылок стало равно 0.

Виртуальная файловая система – прослойка между пользователем и ОС для взаимодействия с файлами. В ней, например, есть каталог `/dev` с виртуальными девайсами (`null, zero, random`). Им не соответствует никакое дисковое пространство.

`swapfile` в Linux — это файл подкачки, специальный раздел на жёстком диске, расширяющий объём оперативной памяти (ОЗУ) в случае, если его не хватает.

Другое определение (другими словами) `swapfile` – файл подкачки – место на диске, куда ОС загружает спящие процессы, когда ей не хватает оперативной памяти.

Пункт 12

Программа `mv` переименовывает файл или перемещает в другую директорию.

Ожидается ответ, что он использует сисколлы `link` и `unlink` поочереди. (На самом деле если мы посмотрим `strace` то он использует сисколл `renameat2` – это знать не обязательно).

`rm` – удаляет жёсткую ссылку. Если их количество равно 0, то удаляет непосредственно файл на диске.

Использует сисколл `unlink`

Пункт 13

Про жёсткие ссылки выше. Там же как создавать оба типа ссылок.

Символическая ссылка – как ярлык в Windows. С такимы ссылками нужно быть аккуратными, так как, если переместить саму ссылку или то, на что она ссылается, то ссылка станет битой (нерабочей). Можно сказать что в ней написан относительный путь к `target`. Хотя так и есть, можно посмотреть содержимое именно самой ссылки с помощью `readlink your-symlink`

С точки зрения ФС, символическая ссылка - особый тип файла. Он содержит только путь к файлу. Имеет место на жёстком диске (так написано в интернетах).

Можно также провести аналогию с указателями и ссылками в плюсах – hardlink как ссылка, объект тот же самый, просто другой псевдоним.

`ln` использует сисколл `link` в случае жёстких ссылок

И сисколл `symlink` для символических ссылок. Подробно читаем в man'е

Пункт 14

В начале каждой строчки вывода команды `ls -l` будут располагаться права доступа к файлу (например, `rwxrwxr-x`):

1. `-` – отсустивие правила
2. `r` – право на чтение
3. `w` – право на запись
4. `x` – право на выполнения
5. Вначале идут права для владельца(user), затем для группы владельцев(group), затем для всех остальных(other) (ugo)

Команда `chmod` помогает сменить права у файла (аналогичный сисколл `chmod`). Как пользоваться `chmod`? Знак + означает добавить правило. Знак - значит удалить правило. Слева от знака (без пробела) пишем для кого меняем правило (`u - user, g - group, o - other, a - all`. Если не написано, то эффект как будто там `a`). Справа от знака (снова без пробела) пишем какое право меняем (`r, w, x`).

В сисколле задаём битовую маску в 8-ричной системе счисления (ex 0567 $\iff$ -r-xrw-rwx);

Про права для директорий написано выше

`chown <username> <filename>` – поменять владельца файла. (Change owner). Использует одноимённый сисколл `int chown(const char *pathname, uid_t owner, gid_t group)` (Ну и перед этим `stat`, чтобы понять, кому файл принадлежал до). Посмотреть поля структурки `stat` можно так `man 3 stat`

`chgrp <groupname> <filename>` – поменять группу владельца файла. (Change group). Тоже использует сисколл `chown`

/* Made by GPT (но это вроде правда)

`suid` (Set User ID) — это специальный бит в системе `Unix/Linux`, который позволяет пользователю запускать исполняемый файл с правами владельца этого файла, а не с правами текущего пользователя. Это часто используется для программ, которым нужны привилегированные права для выполнения определённых действий.

`s` — когда `suid`-бит установлен, и файл исполняемый. Этот символ в праве доступа указывает на то, что при выполнении файла процесс будет иметь права владельца файла.

`S` — это также `suid`-бит, но если файл не является исполняемым. Он указывает на то, что `suid`-бит установлен, но выполнение файла невозможно, так как он не имеет разрешения на выполнение.

Например, утилита `passwd` для изменения пароля пользователя. Она должна иметь `suid`, чтобы обычные пользователи могли изменять свои пароли, но не имели права изменять файлы системной учетной записи.

`sudo` — программа, которая позволяет пользователю выполнять команды от имени другого пользователя (обычно суперпользователя). `suid` даёт права на выполнение привилегированных команд.

Выставить `suid`-бит можно с помощью утилиты `chmod` – `chmod +s filename` (Оно выставляется вместо `w`)

*/

Пункт 15

Все процессы можно посмотреть по пути `/proc/`

Это директории (виртуальные), в которых содержится вся информация о процессе. В том числе содержит поддиректорию `/fd/` – с помощью ls -l fd можно из терминала посмотреть, какие файловые дескрипторы сейчас открыты у данного процесса и какие файлы им соответствуют. Или использовать следующую утилиту

`lsof` – утилита, отображающая открытые файлы процесса (на самом деле читает соотвествующие

виртуальные файлы).

Для демонстрации можно в одном терминале запустить `cat < /dev/zero`, в другом `lsof -p $(pgrep cat)` (или сразу `pid`) (Не забудьте после демонстрации выключить первый процесс.)

Чтобы посмотреть из терминала, какие процессы сейчас держат открытым данный файл можно также поспользоваться командой `lsof | grep /dev/zero` (Вообще у меня сработало и без `grep`. Просто `lsof /dev/zero`)

Или можно посмотреть командой `fuser file_name` (Взято со StackOverflow)

Чтобы реализвовать это на `C` с помощью сисколлов надо:

- Зайти в директорию `proc` с помощью `open` (Наверно всё же надо использовать библиотечную функцию `opendir`)
- Начать проходить по всем процессам с помощью библиотечной функции `readdir`. (Наверное сисколл `read` – он по идее должен возвращать имена файлов. Кому не в падлу, проверьте)
- Найти там директорию `/fd/`. Пройтись по ней, беря информацию о файле с помощью сисколла `stat`

Пункт 16

Блочные и символьные устройства. Разница в том, что символьные устройства возвращают результат посимвольно (`read` считывает последовательно по одному символу).

Блочные посылают порциями фиксированного размера.

Все устройства лежат в `/dev`. Пример символьного – `/dev/rand`, блочного – `/dev/nvme0n1p1` – первый раздел жёсткого диска (можно убедиться с помощью `df -T -h`) (вероятно, посылает данные по четырёхкилобайтным страницам).

Чтобы прочитать данные с какого-нибудь символьного устройства, а также отправить данные на устройство можно использовать известные сисколлы `open/close, read/write`
(по идее, иначе зачем всё это нужно?)

Пример виртуальных устройств:

- `/dev/zero` – Поток нулевых байт. Можно читать из него бесконечно.
- `/dev/random` – Поток рандомных байт. Можно читать из него бесконечно.
- `/dev/null` – Бездна. Всё что туда запишешь пропадёт.

Монтирование файловых систем – системный процесс, подготавливающий раздел диска к использованию операционной системой. (присоединение)

Для монтирования и отмонтирования есть команды `mount` и `umount` (и такие же сисколлы).

Синтаксис `sudo mount device mountpoint`

`mountpoint` Должен уже быть по умолчанию (мб есть флаг который создаёт).

Как смонтировать куда-то? (я даже флешку нашёл для гайда)

1. Вставляем влешку и смотрим командой `df` куда она смонтирована. У меня такая строчка:
   `/dev/sda1 23664832 5445664 118219168 5% media/kirill/Windows USB`
2. Для начала отмонтируем. В моём случае команда `sudo umount /dev/sda1`
   Можем убедиться командой `df`, что устройство пропало
3. Создадим директорию, куда вмонтируем дальше: `mkdir ~/myusb`
4. Монтируем `sudo mount /dev/sda1 ~/myusb`
5. Убеждаемся `df` что всё смонтировалось куда нужно:
   `/dev/sda1 123664832 5445664 118219168 5% /home/kirill/myusb`

Сисколы тоже `mount/umount` (и куча других, вспомогательный. Подробнее используйте `strace`).

Раздел 3. Память

Пункт 17

/* Крайне рекомендуется посмотреть 7 лекцию. Там всё нормально объяснено. И/или конспект Дениса. */

Виртуальная память – иллюзия, предоставленная операционной системой для процесса. Мы привыкли, что обращаемся по каким-то адресам красивым (массивы неприрывны, как и секции `text, data, stack`). На самом деле это всё неправда. Между настоящим `RAM` и нашим процессом существует прослойка – виртуальная память (виртуальная адресация), поддерживаемая самой ОС. Чтобы посмотреть виртуальные адреса процесса, нужно посмотреть на содержимое файла `maps` – `cat /proc/$(pgrep a.out)/maps`

Зачем это нужно:

- Полное изолирование процессов друг от друга. Такой подход исключает возможность обращения не к своей памяти напрямую
- Удобная абстракция. К примеру того, что массив лежит непрерывно. На самом деле он может располагаться в оперативке кусками (выделять непрерывный кусок `RAM` – не всегда получится, а так у нас есть абстракция что всё хорошо)
- Задавать различные права доступа к различным кускам памяти. Например, может существовать неизменяемый участок памяти, видный нескольким процессам.

Вся память поделена на страницы (обычно по 4Кб), для удобства их хранения и скорости доступа к ним. То есть ОС оперирует не несколькими байтами, а страницами.

Таблица страниц — это структура данных, используемая системой виртуальной памяти в операционной системе компьютера для хранения сопоставления между виртуальным адресом и физическим адресом.

/* Копи-паста с конспекта Дениса

Page tables (maps) хранятся внутри RAM и устроены следующим образом:

1. 52 бита адреса разбиваются на 4 равных куска по 13 бит.
2. Имеет структуру B-дерева.
3. В каждой вершине этого дерева всевозможные конфигурации этих кусков из 13 бит (2^{13} возможных вариантов).
4. Получается дерево высоты 4, в котором на каждом уровне определяется наличие определенного последовательного куска длиной 13 бит.

5. На последнем уровне находятся действительные физические адреса в RAM (или они могут отсутствовать).

Механизм виртуальной памяти работает следующим образом:

1. Процессор хочет обратиться к какому-то виртуальному адресу и получить его содержимое (если получает, то сохраняет себе в `cache`, чтобы лишний раз не ходить).
2. С этим запросом он обращается в **MMU** (Memory Management Unit), который располагается рядом с частью процессора, исполняющей код.
3. MMU содержит в себе специальную map структуру (`TLB cache`), которая хранит отображение из виртуальных адресов в физические в RAM.
4. Напрямую к RAM не обращаемся, существует специльная прослойка (шина) для доступа к ней.
5. Если запись с нужным виртуальным адресом находится, то используется она, иначе совершается так называемый **page walk**, после которого если адрес был найден, то он сохраняется в MMU, если и в ходе него не удалось найти нужный адрес, то возникает **page fault**.
6. Если случается page fault, то в дело вступает операционная система (начинается исполнение кода ядра операционной системы, текущий процесс останавливается). На самом деле операционная система выделяет физическую память на виртуальную память лениво. Поэтому page fault мог означать, что нужная память еще не была выделена. Соотвественно, если OS обещала память, то обновляется maps и отдается соотвествующий адрес, в противном случае – **SEGFAULT**, процесс аварийно завершается.
7. RAM хранит помимо основной информации также информацию с **maps**, которые были рассмотрены выше, именно сюда мы приходим во время page walk.
8. Все рассмотренные выше структуры хранят также права на файлы, и если мы нарушаем эти права, то приходит OS и выкидывает `SEGFAULT`, процесс завершается.

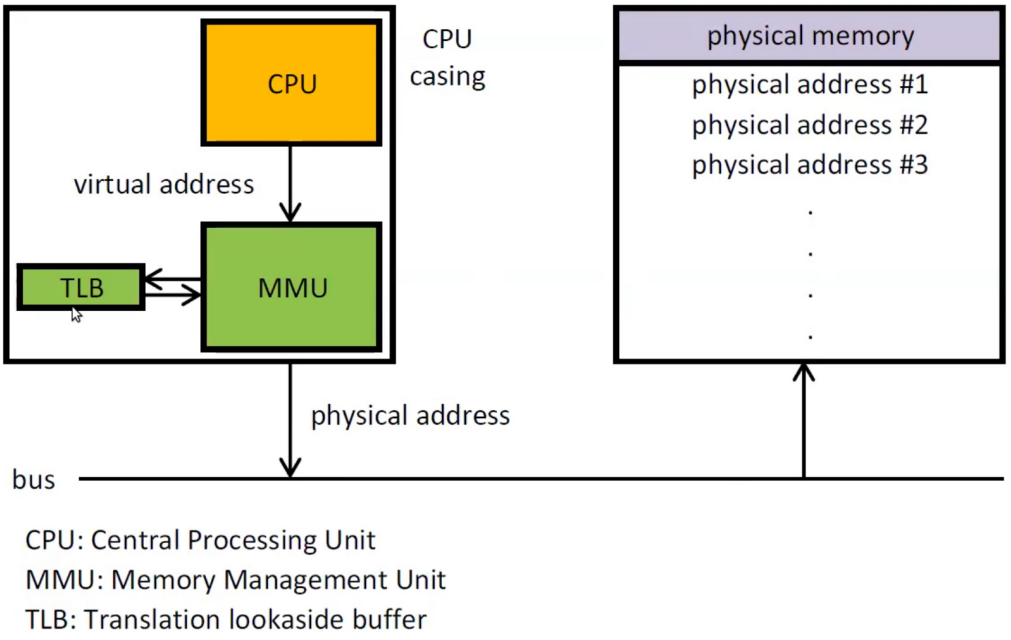

Рис. 1: Наглядное изображение вышесказанного о виртуальной памяти

*/

Отличие `minor` от `major page fault` (на лекции не было, взято из википедии):

- `minor page fault`, означает, что требуемая страница либо уже находится в оперативной памяти, но не отмечена в блоке управления памятью как загруженная, либо она вновь выделена и ещё ни разу не использовалась. Обработчик исключения в операционной системе должен только добавить запись в `Page Tables` для требуемой страницы и указать, что она загружена в память. Загрузка страницы с диска не требуется.
- `major pahe fault` является основным механизмом, используемым операционной системой для выделения программам памяти по их запросу. Операционная система откладывает загрузку частей программы с диска до тех пор, пока программа не попытается получить доступ к ним и тем самым сгенерирует отказ страницы. Если страница не загружена в память на момент отказа, тогда отказ называется значительным или аппаратным. Обработчик отказа страницы в операционной системе должен найти свободное место в оперативной памяти: свободную либо занятую страницу. Занятая страница может принадлежать другому процессу. В этом случае операционная система должна выгрузить данные этой страницы на диск (если они не были выгружены ранее) и пометить эту страницу в таблице страниц процесса как отсутствующую в памяти. Как только свободное место становится доступным, операционная система может загрузить данные для новой страницы в память, добавить её физический адрес в таблицу страниц исходного процесса и пометить страницу, как находящуюся в памяти.

Пункт 18

Адресное пространство процесса делится на:

- `stack` – растёт сверху вниз (от больших адресов к маленьким). Хранятся переменные, адреса возвратов функций и тп
- `heap` – куча. Большой участок памяти между стеком и `text`. Хранятся данные, выделенные с помощью `malloc` (о том, как он работает, будет позже)
- `data` – переменные, зашитые в бинарь. Делится на:
  - `bss` – статические переменные, которые будут проинициализированы `runtime`
  - `data` – проинициализированные статическая переменные программы
  - `rodata` – read-only data, то же самое что и (2) только нельзя изменять (например литерал строки в коде)
- `text` – код программы (можно в `maps` найти по выделенному праву на исполнение `x`)

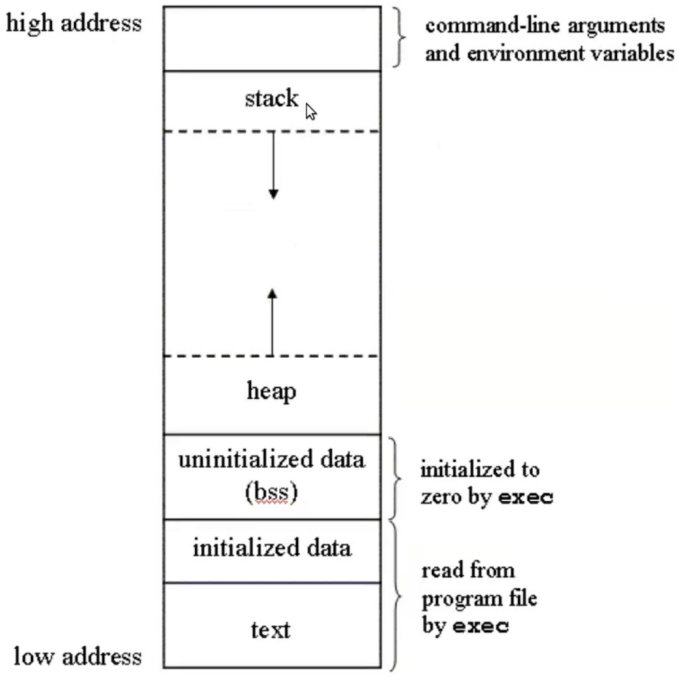

Рис. 2: Адресное пространство программы

`int brk(void *addr)` и `void *sbrk(intptr_t increment)` изменяют местоположение `program break` (они нужны для аллокации памяти в `heap`), которое определяет конец сегмента данных процесса (т.е. `program break` является первым местоположением после окончания неинициализированного сегмента данных). Увеличение `program break` программы приводит к `allocating memory` процессу; уменьшение `program break` освобождает память.

`brk` устанавливает для конца сегмента данных значение, указанное с помощью `addr`, если это значение приемлемо, в системе достаточно памяти и процесс не превышает максимальный размер данных (см. `setrlimit(2)`).

Функция `sbrk` увеличивает объем данных программы на несколько байт. Вызов функции `sbrk` с шагом 0 можно использовать для определения текущего местоположения `program break`.

`mmap` тоже сисколл, необходимый для аллокации памяти в `heap`. Но он выделяет целые страницы и выделяет сегмент с како-то адреса (по умолчанию выбирает его по своему усмотрению). `munmap` – обратное действие к `mmap`, деаллоцирует память.

Пример использования `mmap` и `munmap` в базовом сценарии:

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

int main() {
    void* ptr1 = mmap(nullptr, 1000, PROT_READ | PROT_WRITE,  MAP_PRIVATE |
       MAP_ANONYMOUS, -1, 0);
    void* ptr2 = mmap(nullptr, 70000, PROT_READ | PROT_WRITE,  MAP_PRIVATE |
       MAP_ANONYMOUS, -1, 0);
    std::cout << "ptr1 address: " << ptr1 << std::endl;
    std::cout << "ptr2 address: " << ptr2 << std::endl;

    int res = munmap(ptr1, 1000);
    if (res == -1) {
        std::cerr << "failed to unmap ptr1. errno value: " << errno << std::endl;
    }

    res = munmap(ptr2, 70000);
    if (res == -1) {
        std::cerr << "failed to unmap ptr2. errno value: " << errno << std::endl;
    }
    return 0;
}
```

Подробнее о каждом аргументе `mmap`:

1. `void addr[.length]` – стартовый адресс, где хотим выделить участок памяти. Стоит `nullptr` – ядро выбирает по своему усмотрению
2. `size_t length` – размер выделяемого участка в байтах (выделяемтся постранично минимальный больший размер)
3. `int prot` – права выделяемого участка. Разрешили чтение и запись
4. `int flags` – определяет видимость участка памяти для других процессов и отношение к файлу (с помощью `mmap` можно подключить к своей программе файл). Базованный набор аргументов `MAP_PRIVATE` – запрещаем другим процесам взаимодействовать с этим куском. `MAP_ANONYMOUS`

– просто кусок памяти, без привязки к файлу, если установлен этот флаг, игнорируется следующий параметр (но рекомендуют ставить его -1)

5. `int fd` – файловый дескриптор. Нужен, когда хотим замапить какой-то файл к себе. Здесь игнорим

6. `off_t offset` – отступ в файле, на который ссылается `fd`. Ставим в 0.

Чтобы посмотреть адресное пространство процесса, нужно посмотреть на содержимое файла `maps` – `cat /proc/$(pgrep a.out)/maps`

Пункт 19

Да, можно. `mmap` предоставляет такой интерфейс. Можно передать конкретный адрес первым параметром.

Почему при обращении за границу массива `segfault` происходит не всегда? Потому что память выделяется с излишком (минимальный содержащий размер, в случае `mmap` - блоки по 4 Кб). Поэтому мы можем писать за границу выделенного массива, но всё ещё в "нашей"памяти.

Пример использования `mremap` (сисколл, расширяет память, если может, иначе реаллоцирует):

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

int main() {
    void* ptr = mmap((void*)0x600000000000, 1000, PROT_READ | PROT_WRITE, MAP_PRIVATE
        | MAP_ANONYMOUS, -1, 0);
    std::cout << "ptr addres: " << ptr << std::endl;
    getchar();

    void* ptr_remapped = mremap(ptr, 1000, 9999, MREMAP_MAYMOVE);
    std::cout << "remapped to addres: " << ptr_remapped << std::endl;
    getchar();
    int res = munmap(ptr_remapped, 9999);
}
```

Вот так можно с помощью mmap загрузить файл в оперативную память:

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

int main() {
    int fd = open("example.txt", O_RDWR);
```

```
9     if (fd == -1) {
10        exit(1);
11    }
12    struct stat sb;
13    if (fstat(fd, &sb) == -1) {
14        exit(1);
15    }
16    char* data = (char*)mmap(NULL, sb.st_size, PROT_READ | PROT_WRITE, MAP_PRIVATE,
          fd, 0);
17    if (data == MAP_FAILED) {
18        exit(1);
19    }
20    printf("%s\n", data);
21    data[0] = 'X';
22    if (msync(data, sb.st_size, MS_SYNC) == -1) {
23        exit(1);
24    }
25    if (munmap(data, sb.st_size) == -1) {
26        exit(1);
27    }
28    close(fd);
29 }
```

Очень важную роль здесь играют флаги. Если установлен флаг `MAP_PRIVATE`, то файл нельзя будет поменять. Процессу будет предоставлена копия файла. Он сможет туда писать, но изменения не скажутся на реальном файле. Если установлен флаг `MAP_SHARED` вместо него, то изменения этой памяти будут изменять и файл.

При этом в реальности файл меняется не сразу. ОС запоминает, что нужно его поменять и когда-нибудь поменяет. Чтобы заставить ОС записать изменения сейчас, используется сисколл `int msync(void addr[.length], size_t length, int flags)`. Первый аргумент – указатель на замапленный файл, второй - сколько байт нужно заапдейтить, третий - в каком режиме обновить. Мы ставим `MS_SYNC` - синхронно, т.е. ждём когда реально обновится. (можно поставить асинхронно).

По поводу него интересная строчка в мане: *Without use of this call, there is no guarantee that changes are written back before munmap(2) is called* =)

Пункт 20

Права доступа к памяти почти такие же как и для файла:

- `r` – разрешено чтение
- `w` – разрешена запись

- `x` – разрешено исполнение (послать эти байтики исполняться процессору)
- последний параметр – p или s – `private` или `shared` этот кусок памяти

`int mprotect(void addr[.len], size_t len, int prot)` – устанавливает права доступа к участку памяти. Первый аргумент – указатель на НАЧАЛО куска памяти (т.е. выровнен по размеру страницы, так корректнее), второй – его длина, третий – какие права хотим установить (например `PROT_READ | PROT_EXEC`).

Как с помощью `mmap` загрузить код из библиотеки в память на выполнекние. Подробный гайд.

1. Пусть у нас есть файл `myfunc.cpp` с таким содержанием:
   `double myfunc(double x)  return x * x;` (почему-то x + 5 не работает)))))
   создадим из него объектный файл `g++ -c myfunc.cpp -o myfunc.o`
2. С помощью `readelf -a myfunc.o` находим смещение секции `text` относительно начала файла (должно быть 40)
3. Компилируем код ниже: `g++ mmap_library.cpp`
4. Запускаем и радуемся: `./a.out myfunc.o 7`

```c
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <unistd.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);
    printf("func(%f) = %f\n", argument, result);
    munmap(addr, st.st_size);
}
```

Ошибка `Illegal intstruction` означает, что процессору послали невалидную команду (неправильные байты) и он не смог выполнить эту инструкцию.

Пример программы на C++, которая приводит к этой ошибке:

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

int main() {
    std::vector<char> v;

    for (int i = 0; i < 1000000; ++i) {
        v.push_back(i);
    }

    std::cout << "vector starts at " << (int*)&v[0] << std::endl;
    std::cout << mprotect(&v[0] - 16, 10000, PROT_READ|PROT_WRITE|PROT_EXEC) << std::
       endl;
    v[0] = 0x33;
    void (*f)() = (void(*)()) &v[0];
    f(); // Illegal instruction (core dumped)!!!
}
```

В коде нужно `-16`, так как `malloc` в начале выделенного куска проставляет его размер, а адрес в `mprotect` должен быть выровнен по размеру страницы.

Пункт 21

/* Копипаста с конспекта Дениса

Изначально `malloc` придумал челик **Doug Lea** (его статья об этом, хорошее описание его реализации). Реализация ниже (`dlmalloc`) – не настоящий `malloc`, настоящая реализация называется `ptmalloc`, которая думает о потокобезопасности, и которая и реализована в стандартной библиотеке.

`Malloc` устроен следующим образом (ориентировочно):

1. Вызывает `sbrk` или `mmap` и пытается сократить количество их вызовов, ведь это дорого каждый раз их вызывать.
2. **MMAP_THRESHOLD** – размер аллокации, начиная с которого вызывается `mmap` (разный в разных системах, например, 128 килобайт вполне себе реальная цифра).
3. `mmap` при выделении памяти оставляет первые 16 бит на размер участка памяти (comment: не `mmap` выставляет, а сам маллок. Ммапу пофигу так-то)

4. В самом начале мы делаем `sbrk` на какое-то внушительное число байт (например на `MMAP_THRESHOLD`) и тем самым обеспечиваем себе некий "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 конечно возможно. То есть наше хранилище лежит в `heap`.

5. Храним некоторый массив двусвязных списков, содержащих свободные "чанки"(отрезки памяти из сегмента выше). Где по каждому индексу будут лежать списки с блоками, размеров от 16 до 512 с шагом в 8 байт (`small chunks`) (для каждой ячейки массива свой размер блоков, содержащихся в списке ячейки), а после, далее размеры чанков будут расти уже экспоненциально, пока не дорастут до `MMAP_THRESHOLD`. (comment: их тоже около 64 штук. Они называются `large chunks`)

6. (comment: это без учёта `fastbins`, о нём ниже) 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.

7. Каждый чанк начинается и заканчивается своим размером, занимающим в начале и в конце чанка по 4 байт (comment: мб 8, в зависимости от реализации) (в добавлении к размерам из пункта (5), то есть не включительно). (comment: Так как размер всегда кратен 8, последние 3 бита свободны. Используем их под нужды самого малока. Например в последнем бите хранится информация, занят ли этот(предыдущий) чанк: 1 – если занят. 0 – свободен)

8. Когда освобождается чанк в этом хранилище, то информация выше о размерах 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 структуру из пункта (5).

9. Если чанк не содержит пользовательскую память, то первые 16 байт в нем (comment: после размера) выделяются на хранение указателей списка свободных блоков такого же размера.

*/

`fastbins` – только что освобождённые чанки (небольшие). То есть перед тем как добавить в структуру, их связывают в отдельный список. (Если вас попросили деаллоцировать сколько-то байт, то вероятно скоро попросят аллоцировать столько же). 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 структуру.

`malloc` использует в основном 2 сисколла – `brk` (`sbrk`) для небольших размеров и `mmap` для значительных запросов. (а вся троица `malloc, realloc, free` использует дополнительно `mremap, munmap`, возможно ещё `mprotect` для выдачи прав).

Раздел 4. Процессы и треды

Пункт 22

Процесс - это программа (исходный код) и некоторая дополнительная информация (маппинг памяти, `pid`, `ppid` и многое другое). Процессы исполняется на процессоре. Но процессов много, а ядер мало. Поэтому они исполняются поочереди, с механизмом `прерывания` (!ВАЖНОЕ СЛОВО). То есть в процессоре есть `handler`-ы прерываний. Когда поступает сигнал к какому-то процессу, процессор переходит в `привилегированный режим` и идёт исполнять код ядра ОС (что именно исполнять захардкожено для каждого сигнала самой ОС). Можно сказать, что приходит сама ОС и разбирается с тем, что "натворил"процессор с этим процессом и решает, кого поставить дальше на исполнение. Сигналы могут быть разными, в том числе, когда ставится процесс, ему может быть выдано время выполнения. По истечение этого времени возникает сигнал. (Процессор это отслеживает с помощью `тикающего устройства`).

Чтобы посмотреть все текущие процессы, используем команду `ps aux` или синоним `ps -ef` – выводит статическую таблицу процессов. Что означают эти флаги, можно не знать (так-то можно посмотреть в `man`). Ещё есть предустановленная утилита `top` и более "современная"`htop` – это уже динамически обновляющиеся таблицы.

`pid` – Process ID, натуральное число, соответствующее этому процессу. У самого первого процесса `pid = 1`. Это так называемый процесс `init`. На лекции он породил процесс-демон `systemd` (технические процессы, работающие в фоне, самой ОС называются демонами) с другим `pid`, который уже запускает остальные процессы. (У меня не так, у меня `systemd` имеет `pid` 1. Тут могут быть различия в зависимости от ОС).

`ppid` – Parent Process ID, id-шник родительского процесса (того, кто породил этот процесс).

Чтобы посмотреть дерево процессов используем команду `pstree`

Чтобы посмотреть потребление памяти, потребление CPU каждым из процессов, набрать любую из вышеперечисленных команд (кроме pstree, лучше `htop`) и посмотреть соответсвующий столбец – MEM и `CPU` соответственно. (`MEM` вообще делится на 2 колонки – `VIRT` и `RES` – то, сколько процессу пообещала ОС и сколько он реально сейчас использует. Напомню, что память выделяется лениво) (А вообще есть директория `/proc/<ppid>/`, там можно найти всю информацию. Посмотреть за что какаая директория отвечает `man proc` – откроется пятый разделел мана)

Прироритет процесса – хорошее объяснение на Хабре:

Любезность или приоритет `nice` (`NI`) – это приоритет процесса в пространстве пользователя, варьирующаяся от -20, что есть самый высокий приоритет, до 19, соответственно наименьший приоритет. Это может запутать, но представьте это именно как любезность, т.е. чем процесс любезнее, тем он уступчивее другим процессам (получит меньше времени исполнения, если будет конкуренция за ресурсы процессора).

Приоритет (`PRI`) же в свою очередь это параметр приоритета в пространстве ядра. Приоритет варьируется от 0 до 139. (Нас интересует от 100 до 139)

Можно изменить любезность процесса, и тогда, возможно, ядро примет это к сведению, но сам приоритет менять нельзя.

Соотношение любезности и приоритета следующее: `PR = 20 + NI`. Таким образом область определения PR = 20 + (-20 to +19) лежит в отрезке от 100 до 139 (в `htop` показывается `PR - 100` для нас).

Узнать `NI` для своего процесса можно с помощью `htop` или командой `ps -l <pid>`

Можно запустить программу сразу с каким-то `nice` (по умолчанию он равен 0) с помощью `nice -n 15 ./a.out` (при этом, если хотим установить отрицательное значение, надо сначала написать `sudo` (подробнее о принципе работы `sudo` будет в следующем пункте)).

Чтобы изменить `nice` уже работающего процесса: `renice -n 15 $(pgrep a.out)` (установили `nice = 15`).

Для работы с `nice` из кода есть одноимённый сисколл: `int nice(int)` - принимает, на сколько нужно увеличить `nice` процесса. В случае успеха возвращает текущий `nice`. (отрицательные значения опять же нельзя писать, если нет соответсвующих привилегий).

`uid` – User ID, имя того, кто запустил текущий процесс (если упростить). Процесс будет обладать праввами этого пользователя.

`euid` – Effective User ID – имя того, с чьими правами исполняется данный процесс.

Поясняю разницу: когда мы просто запускаем обычный процесс, он наделяется правами того, кто его запустил, то есть `uid = euid`. Но иногда мы хотим изменить свои права (права можно ТОЛЬКО ослабить. Нельзя наделить себя правами, которыми ты не обладаешь) на время. Например, мы запустили программу из-под `root`. Тогда `uid = 0, euid = 0`. Мы временно хотим себе запретить уменьшать `nice`, получать доступ к системным файлам и т.п. Но при этом потом хотим вернуть себе эти права. Тогда мы меняем `euid` и программа начинает исполняться с другими правами (`uid = 0, euid = 1000`). Но затем мы можем обратно вернуть себе права `root`, так как `uid` всё ещё 0 (у нас есть данные права).

Сисколлы, которые помогают с этим работать: `getuid` и `geteuid` чтобы узнать текущие `uid и euid`. `setuid` и `seteuid` - чтобы их установить.

`cwd` – Current Working Directory, это то, что программа считает точкой "."(изначально откуда она была запущена). Узнать изнутри программы – сисколл `char* getcwd(char buf[.size], size_t size)`. Изменить – `int chdir(const char *path)`. Чтобы узнать `cwd` для какого-то процесса из терминала – `sudo ls -l /proc/$(pgrep a.out)/cwd` (или `readlink /proc/$(pgrep a.out)/cwd`, ведь `cwd` это лишь символическая ссылка).

Пункт 23

`int fork()` - сисколл, раздваивающий вашу программу (буквально. Он полностью копирует стек, маппинги (тут Мещерин сказал, что с нюансами, а с какими - не сказал)). Возвращает РОДИТЕЛЮ `pid` ребёнка, а РЕБЁНКУ - 0 (он-то может узнать `pid` родителя с помощью `getppid`). Важно понимать, что буффер `cout` тоже копируется, поэтому в следующем коде "Hello"выведется дважды (вопрос на уд):

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

int main() {
    std::cout << "Hello! ";
    int pid = fork();
    if (pid == 0) {
        std::cout << "I was born!" << std::endl;
    } else {
        std::cout << "I am parent!" << std::endl;
    }
}
```

`exec` (сисколл называется `execve` так-то) – запускает вместо вашего процесса другую программу (которую вы указали). При этом `она будет иметь ваш pid`, и ваши права (что логично). Как раз так и работает `sudo`! Он делает `execve` на вашу прогу (перед этим требует пароль, конечно). А как `sudo` сам получает права рута? За счёт `suid`-бита! Подробнее в Пункте 14. Подробнее про параметры `int execve(const char *pathname, char *const _Nullable argv[], char *const _Nullable envp[])`:

1. Вроде понятно – путь к исполняемой программе
2. Аргументы, которыми запустить программу. По соглашению, первым элементом должно быть само название программы. Далее идут сами аргументы. И массив должен заканчиваться на `NULL`.
3. Переменные окружения в форме `key=value`, которые передать программе (например LD_PRELOAD=. или LD_LIBRARY_PATH=/lib/). Тоже должен заканчиваться на `NULL`.

Существуют разные версии `exec` (можно посмотреть `man 3 exec`). Отличие их от `execve`:

- `execl(const char* pathname, const char *arg, ... /*, (char *) NULL */)` – отличие в том, что принимает только аргументы программы, причём как `variadic` (`arg[0]` по-прежнему должно быть само название программы). Последним аргументом должен быть `(char*)NULL` (нулевой указатель, скастованный к `char*`)

- `int execlp(const char* file, const char *arg, ... /*, (char *) NULL */)` – как и предыдущий пункт, только принимает первым аргументом название исполняемого файла (не путь). Он ищет его в директориях, указанных в `env` переменной `PATH` (по умолчанию ищет в `"/bin:/usr/bin"`). При этом в `file` не должно быть слешей (иначе PATH игнорируется)
- `int execle(const char *pathname, const char *arg, ... /*, (char *) NULL, char *const envp[] */)` – тоже самое, что и `execve`, только все данные передаются как `variadic`. Параметр `arg` отделяется от `env` с помощью аргумента `(char*)NULL` (после него считается, что идут параметры `env`). Правила те же.
- Дальше идут ещё 3 типа – `execv, execvp, execvpe`. Глобальных отличий от предыдущих вариантов нет, параметры имеют то же значение.

Просто интуиция, что означают эти буковки, после `exec`:

- `v` – `variables`. Возможность передать `argv` новой программе
- `e` – `enviroment`. Возможность передать `env` новой программе
- `l` - как и `v`, только передаём аргументы как `variadic`.
- `p` – `path`. Использовать переменную `PATH` для поиска исполняемой программы. Имеется ввиду, что нам передали только `filename`

В чем необычность функций `fork` и `exec`, что происходит при их вызове? (хз что он тут хочет услышать) Вероятно то, что это не функции в обычном понимании. Они меняют процесс полностью (в первом случае раздваивает, во втором меняет программу на другую). Что происходит при вызове `fork`, написал вначале. При вызове `exec` – отличная строка в man: « execve() не возвращает результат в случае успеха, и text, initialized data, uninitialized data (bss) и stack вызывающего процесса перезаписываются в соответствии с содержимым вновь загруженной программы, Memory mappings не сохраняются » (можно почитать ещё в `man 2 execve`)

Пример вызова из программы другой программы, используя `fork+exec`:

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

int main() {
    int pid = fork();
    if (pid != 0) {
        std::cout << "Parent pid: " << getpid() << " child pid: " << pid << std::endl
           ;
        int status;
        int recieved_pid = wait(&status);
```

```
    } else {
        char* argv[] = { "/usr/bin/ls", "-l", ".", NULL };
        char* env[] = { NULL };
        int error = execve("/usr/bin/ls", argv, env);
        std::cout << "I'm a children with pid: " << getpid() << std::endl;
    }
}
```

Объяснение: изначально раздваиваем нашу программу форком. Если мы родитель (`pid` != 0), то выведем информацию о программе и дождёмся сына с помощью функции `wait` (её нет в билетах, но я думаю нужно знать что это такое – функция дожидается, пока завершится какой-нибудь из порождённых нами детей (на самом когда ребёнок изменит своё состояние. О состояниях позже). Она принимает указатель, куда положит результат, с которым завершился сын и вернёт `pid` завершённого сына. Есть ещё `waitpid` чтобы подождать конкретного ребёнка).

Если мы ребёнок – запустим программу `ls -l .`. ВАЖНО! Если всё прошло хорошо, то мы из `exec` не вернёмся и `cout` с информацией о ребёнке НЕ выведется.

`fork-бомба` – вредоносная программа, которая в бесконечном цикле вызывает `fork`, тем самым нагружая процессор и используя ресурсы программы. Самая простая `fork-бомба`:

```
#include <unistd.h>
int main() {
    while(true) {
        fork();
    }
}
```

Про `fork` ещё спросят что будет, если ребёнок завершится раньше родителя. Так вот, тогда сын перейдёт в состояние `Z` (в следубщем пункте), так как родителю ещё возможно нужно будет получать мета-информацию о своём ребёнке

Пункт 24

В выводах команд aka `ps` или в `htop` можно посмотреть статус процесса, которые бывают следующие (перечислены внизу, если ввести команду `man ps` – можно быстро найти секцию, если уже внутри мана ввести / (слеш, означает поиск), `STATE` (слово, по которому ищем) и нажать `enter` (самое время рассказать как пользоваться man-ом =)):

1. `D` – непрерываемый сон (процесс ожидает некое событие, не реагируя при этом ни на какие сигналы). Например, если процесс долго читает с диска, то ОС запрещяет как-либо его трогать. (Искуственно воспроизвести сложно, команд не будет)
2. `I` – Незанятый поток ядра.

Если повезёт, то в `htop` в колонке `S` будет процесс с таким состоянием.

3. `R` – исполняемые в данный момент, или стоящие в очереди на исполнение.
   Запускаем программу с `while(true)` и смотрим в `htop`, там он появится

4. `S` – спящие процессы (чего то ожидает, например ввода с клавиатуры). Прерываемый сон.
   Можно наблюдать в `htop`

5. `T` – остановленные процессы. Важно понимать разницу между `S` и `T`. В первом случае процесс просто ожидает какого-то результата и его можно просто убить командой `kill <pid>`.Во втором процессу был послан сигнал `SIGTSTP` (посмотреть все сигналы `man 7 signal` но о ни будет позже). А значит процесс возобновит свою работу только когда ему придёт сигнал, разрешающий возобновление работы (типа `Cont`). При этом в этом состоянии процесс не может быть убит командой `kill <pid>`. Нужно использовать более жёсткий вариант: `kill -KILL <pid>` или то же самое `kill -9 <pid>`

6. `X` – мёртвый процесс (такое состояние невоможно увидеть)

7. `Z` – "зомби", завершившиеся дочерние процессы (у них очищены ресурсы, память в том числе), которые еще не "подобрал" их родитель (не был вызван на них `wait`).

Приостановить процесс – `Ctrl + Z` в окне с работающем процессом или `kill -STOP <pid>`. Чтобы его возобновить, можно воспользоваться командой `kill -CONT <pid>` (для примера используем программу с `while(true)`).

`bg` и `fg` – команды для работы с фоновыми процессами.

`./a.out &` – запустить процесс в фоне. Когда запускается как обычно, то на самом деле терминал ждет завершения процесса системным вызовом `wait`.

`jobs` – команда, отображающая процессы, запущенные в фоне для этого терминала. Если их несколько, то им присваиваются порядковые номера.

`fg` – команда, выводящая из фонового режима последний процесс из списка, выводимого программой выше (помеченный +).

`bg` – команда, запускающая в фоновом режиме последний процесс из списка выше (помеченный +), который ранее был остановлен.

Посмотреть состояние процесса можно командой `ps -l | grep a.out` Второй столбец - состояние процесса. Или вспоминаем про директорию `/proc` – `cat /proc/$(pgrep a.out)/stat` – после названия будет указан статус процесса. Ну или используем любимый `htop`

Пункт 25

`rlimit` для процесса — это ограничивающая граница на потребление процессом различных системных ресурсов. Текущие лимита процесса можно посмотреть в файле командой `cat /proc/$(pgrep a.out)/limits`

Про `getrlimit` и `setrlimit`: Оба сисколла используют структуру

```
struct rlimit {
    rlim_t rlim_cur;  /* Soft limit */
    rlim_t rlim_max;  /* Hard limit (ceiling for rlim_cur) */
};
```

`soft limit` - это значение, которое ядро устанавливает для соответствующего ресурса. `hard limit` действует как верхний предел для `soft limit`: непривилегированный процесс может устанавливать только для своего `soft limit` значение в диапазоне от 0 до `hard limit` и (необратимо) понижать свой `hard limit`. Привилегированный процесс может вносить произвольные изменения в любое из предельных значений.

`int getrlimit(int resource, struct rlimit *rlim)`:

- Первый аргумент – то, что мы хотим увидеть (поменять). Перечислены в `man 2 getrlimit`, например `RLIMIT_STACK` чтобы запрсить себе другой размер стека.
- Второй аргумент – указатель на структурку выше, куда будет записан ответ

`int setrlimit(int resource, struct rlimit *rlim)`:

- Первый аргумент – то, что мы хотим поменять.
- Второй аргумент – указатель на структурку выше, откуда будет взято значение для изменения. Пишем то, что хотим изменить в `rlim_cur`

Вот так можно запросить себе больший размер стека, чем дан изначально (считаем что мы просим всегда больше, чем сейчас, так- там ифец можно поставить)

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

int main() {
    const rlim_t kStackSize = 210 * 1024 * 1024;
    struct rlimit rl;
    int result;

    result = getrlimit(RLIMIT_STACK, &rl);
    if (result != 0) {
```

```
        std::cerr << "Failed getrlimit\n";
        exit(1);
    }

    std::cout << "Curr soft limmit: " << rl.rlim_cur << "\n";
    rl.rlim_cur = kStackSize;
    result = setrlimit(RLIMIT_STACK, &rl);
    if (result != 0) {
        std::cerr << "Failed setrlimit\n";
        exit(1);
    }
    std::cout << "Limit set to: " << kStackSize << "\n";
}
```

Установить процессу ограничение на использование памяти и/или процессорного времени можно точно так же, только первый аргумент `RLIMIT_DATA` (это вся память, включая `heap, data, text...`) и `RLIMIT_CPU` (в секундах) соответсвенно.

Если привысить данные лимиты, то в первом случае сисколлы, выделяющие память (`brk, sbrk, mmap`) упадут с ошибкой `ENOMEM`, а во втором случае будет послан сигнал `SIGXCPU`. Но он может быть пойман процессом, поэтому процессу он будет посылаться каждую секунду, пока процесс не привысит `hard limit` – тогда будет послан `SIGKILL`. Всё это взято из man-а: `man 2 getrlimit`

Пункт 26

`seccomp` – динамическая библиотека, с помощью которой можно установить какое-то действие при вызове какого-либо сисколла (Например, завершить программу или чтобы сисколл возвращал ошибку), также можно указать параметры, с которыми нельзя вызывать сисколл.

Установить библиотеку – `sudo apt install libseccomp-dev`. Скомпилировать программу: `g++ seccomp.cpp -lseccomp`

Пример программы с запретом вызова сисколла `fork` и `execve` (Очень забавно. `fork` – это не сисколл в Си, это стандартная функция, вызывающая сисколл `clone`, НАС ОБМАНЫВАЮТ, взято со StackOverflow):

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

void setup_seccomp() {
    scmp_filter_ctx ctx = seccomp_init(SCMP_ACT_ALLOW);
    if (ctx == nullptr) {
        std::cerr << "Failed to init seccomp\n";
        exit(1);
```

```
    }
    seccomp_rule_add(ctx, SCMP_ACT_ERRNO(EPERM), SCMP_SYS(clone), 0);
    seccomp_rule_add(ctx, SCMP_ACT_KILL_PROCESS, SCMP_SYS(execve), 0);

    if (seccomp_load(ctx) < 0) {
        std::cerr << "Failed to load seccomp\n";
        exit(1);
    }

    seccomp_release(ctx);
}

int main() {
    setup_seccomp();
    std::cout << "Setted up seccomp\n";

    int res = fork();
    std::cout << "Fork res: " << res << " errno status: " << errno << "\n";

    char* argv[] = { "/usr/bin/ls", "-l", ".", NULL };
    char* env[] = { NULL };
     res = execve("/usr/bin/ls", argv, env);
    while(true) {}
}
```

Объяснение:

- Изначально разрешаем все действия и инициализируем фильтр командой `seccomp_init(SCMP_ACT_ALLOW)`
- Далее добавляем правила (фильтры) для `fork` (clone) командой `seccomp_rule_add(ctx, SCMP_ACT_ERRNO(EPERM), SCMP _SYS(clone), 0);`
  - Первый аргумент – сам контекст, куда записываем правила
  - Второй – действие при вызове сисколла. Здесь установлена команда "Завершись неудачно и выставь значение `errno` в флаг `EPERM` (нет доступа)". Подробнее что там можно поставить – `man 3 seccomp_rule_add`
  - Третий – такая конструкция с названием сисколла
  - Четвёртый – аргументы, с которыми запускается сисколл (в плане будем ловить только то, что вызывается с такими аргументами, например можно запретить вызывать `write` на 3 `fd`)
- Аналогично для `execve`, только выставлена реакция "Убить процесс". Если будет вызван сисколл с такой реакцией, то программа завершится аварийно с ошибкой `Bad system call (core dumped)`

- Далее загружаем инструкции и говорим следовать им.
- После всего этого программа будет работать бесконечно, так как `execve` и `fork` не вызовутся

Пункт 27

Сигналы – один из видов `IPC` – `Inter Process Communication`. Всего этих видов глобально 3: (`signal, pipe(FIFO), shared memory`)

Когда приходит сигнал какому-то процессу, случается прерывание и процессор идёт исполнять код из `handler` этого сигнала.

Чтобы послать сигнал процессу из терминала используем команду `kill -SIGNALNAME <pid>`. Из кода программы одноимённый сисколл `kill(getpid(), SIGNALNUM)` (принимает кому послать сигнал (если указать -1, то оно пошлёт этот сигнал всем =), кому может в соответствии со своими правами) и какой сигнал в виде целочисленной константы (у каждого сигнала свой номер))

Сигналы делятся на 5 видов по реацкии на них (посмотреть в `man 7 signal`):

- `Term` – стандартное действие закончить процесс, умереть (Пример `SIGKILL` когда вызываем `kill -KILL $(pgrep a.out)`)
- `Ign` – стандартное действие игнорировать данный сигнал. (Нужно для передачи какой-то информации. Например `SIGCHLD` посылается родителю, когда ребёнок завершается)
- `Core` – как `Term`, только ещё и записать состояние ядра (`core dump`. Например, наш любимый `SIGSEGV` - сегфолт).
- `Stop` – остановить процесс. (например сигнал `SIGTSTP` посылается при нажатии `Ctrl + Z`).
- `Cont` - продолжить процесс, если он был остановлен. (например сигнал `SIGCONT` - мы его использовали уже. Кстати, он единственный их этого раздела)

Ещё интересный сигнал `SIGHUP` – запрос протух (трубку положили). Чтобы изебжать такого сигнала, можно запускать программу с помощью `nohup` (утилита)

Я думаю ему хватит этих сигналов. Если что открываем `man` и ищем знакомые буквы. Стандартную реакцию я тоже описал.

Вручную из терминала вызвать у стороннего процесса `segfault` – `kill -SIGSEGV $(pgrep a.out)`

Вот так можно в коде программы послать сигнал самому себе / заснуть до прихода сигнала (нужное раскомментировать) (строчку с `kill` можно заменить на `raise(SIGSTOP)` – посылает сигнал самому себе. Просто функция-обёртка). `pause` – приостановить программу до прихода любого сигнала:

```cpp
#include <signal.h>
#include <iostream>
#include <unistd.h>
int main() {
    std::cout << "I wanna sleep..." << std::endl;
    // kill(getpid(), SIGSTOP);
    pause();
    std::cout << "Hello, I just wake up =)" << std::endl;
}
```

Есть всего 2 сигнала, которые нельзя перехватить (заблокировать или установить кастомное действие) – `SIGKILL` и `SIGSTOP`.

Пункт 28

Чтобы сделать кастомный обработчик сигналов нужно написать совй `handler` и воспользоваться сисколлом `signal` (`man 2 signal`). Пример, как из кода программы перехватывать segfault и делать что-то нестандартное при его наступлении:

```cpp
#include <signal.h>
#include <iostream>
#include <unistd.h>
int* p = NULL;
void handler(int signum) {
    std::cout << "Cought signal with num " << signum << std::endl;
}
int main() {
    signal(SIGSEGV, &handler);
    *p = 5;
}
```

Пару важных комментариев к коду:

- `handler` принимает номер сигнала который пришёл. Ничего не возвращает. В `signal` первым параметром пишем номер сигнала, который хотим обработать, вторым - указатель на функцию-обработчик.
- Данный код будет выполняться бесконечно и ловить `SIGSEGV` так как после выполнения обработчика процессор возвращается к той же ассемблерной инструкции, на которой словил сигнал. При этом (ВАЖНО) если изменить `p` внутри обработчика и сделать валидным поинтором НИЧЕГО не изменится (по причине выше. Ассемблерная инструкция осталась та же).
- Есть пример, когда из обработчика можно починить `SIGSEGV`. Пример связан с `mprotect` (попытались записать туда, где нет прав. В обработчике выдали себе права). Код можно посмотреть

в 12 лекции (тайминг 1:11:52).

- Данный способ обработки сигналов считается устаревшим, есть более удобные библиотеки.
- Вот то что мы делали в обработчике – это `UB`, так как `std::cout` не `signal-safety` (такие функции, в которые можно войти ещё раз, пока они исполняются, без возможных проблем. `cout` такой не является из-за буфера). Посмотреть все `signal-safety` функции и почитать поподробнее: `man 7 signal-safety`

Если во время обработки приходит такой же сигнал (который обрабатывается), то он игнорируется (Не может быть получен ещё раз). Если приходит отличный от текущего сигнал, то его обработчик запускается `поверх` текущего, на том же стеке (можно выводить адрес принимаемого номера и в этом убедиться). Но так будет только 1 раз. Потому что ОС запоминает, какие сигналы сейчас в обработке (у неё есть битовая маска). И если поступят такие же сигналы, то ОС запомнит что их нужно послать, дождётся обработки предыдущих таких же и пошлёт снова (короче смотрите лекцию 12, тут нормально тяжело объяснить, тайминг 1:22:00).

Но StackOverflow: Если SIGBUS, SIGFPE, SIGILL или SIGSEGV генерируются, когда они заблокированы, результат не определен, если только сигнал не был сгенерирован kill(2), sigqueue(3) или raise(3).

А вообще, как мы видим, вызывается при втором летящем `SIGSEGV` стандартный обработчик.

Но кончено через `signal` никто сигналы не обрабатывает. Для этого используют `sigaction`. Пример кода, который блокирует получение других сигналов во время обработки какого-либо сигнала:

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

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

int main() {
    struct sigaction sa;
    sa.sa_handler = &handler;

    sigfillset(&sa.sa_mask);
    sa.sa_flags = 0;

    if (sigaction(SIGINT, &sa, NULL) == -1) {
        perror("Failed sigaction");
        exit(1);
```

```
    }
    sleep(5);
}
```

Самое важное наверное здесь `sa_mask` и `sigfillset`. `sa_mask` отвечает за то, какие сигналы не могут поступать во время действия обработчика. То есть если мы заполняем `sa_mask` с помощью `sigfillset`, то никакие другие сигналы не поступят, пока выполняется хендлер.

Что происходит, если сигнал приходит во время выполнения какого-либо сисколла? – На лекции о таком не говорилось, нашёл ответ на StackOverflow: Многие системные вызовы сообщают о коде ошибки `EINTR`, если сигнал поступил во время выполнения системного вызова. На самом деле никакой ошибки не произошло, о ней просто сообщается таким образом, потому что система не может автоматически возобновить системный вызов.

То есть сисколл прерывается и выставляет `errno` в эту ошибку.

Пункт 29

`pipes` – Трубы. Второй вид `IPC`. Имеет два конца: на чтение и на запись. То есть кто-то может читать из него, а кто-то писать.

Особенности общения между процессами с помощью `pipe`:

- Чтение/запись буферизуется ядром
- Если процесс начинает чтение из пайпа до того, как кто-то туда начал писать, он зависает (ждёт ввода)
- Если процесс начинает читать из пайпа, когда конец на запись закрыт (больше никто в него писать не может), то `read` завершается неудачно и процесс продолжает своё исполнение
- Если процесс начинает писать, когда из пайпа никто не читает, то его убивают сигналом `SIGPIPE` – `Broken pipe` (стандартная реакция – `Term`)

Для создания пайпа есть сисколл `pipe(int pipefd[2])` – принимает на вход указатель на массив из 2х интов. В `pipefd[0]` он положит конец на чтение. В `pepefd[1]` – на запись (по аналогии со стандартными файловыми дескрипторами).

Пример кода общения между детьми с помощью `pipe`:

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

```
#include <errno.h>

int main() {
    char buf[1000];
    int pipefd[2];
    pipe(pipefd);

    int pid = fork();
    if (pid == 0) {
        close(pipefd[1]);
        while (read(pipefd[0], &buf, 1) > 0) {
            write(1, &buf, 1);
        }
        close(pipefd[0]);
        write(1, "Child closing\n", strlen("Child closing\n"));
        exit(0);
    } else {
        close(pipefd[0]);
        sleep(1);
        write(pipefd[1], std::string("Hello!").data(), 6);
        close(pipefd[1]);
        wait(NULL);
        exit(0);
    }
}
```

Здесь родитель пишет в пайп, а ребёнок читает.

Чтобы реализовать аналог оператора | (`pipe`) в `bash` нужно сделать открыть пайп. Отдать один конец левому процессу вместо стандартного вывода (сисколл `dup2`), а второй конец другому, вместо стандартного ввода. Сделать fork и `exec`.

Пункт 30

`fifo`-файлы – аналог `pipe`. Это виртуальный файл (не лежит на диске, поддерживается ОС). Из него можно читать и писать.

Особенности:

- Если начать писать в него (и никто до этого не открывал его на чтение), то ввод заблокируется до тех пор, пока кто-то не откроет его на чтение
- Чтение работает аналогично как в пайпе – блокируется пока кто-то не начнёт писать
- Если все читатели файла закрылись в процессе записи в фифо, то процессу посылают сигнал `Broken pipe` и он умерает.

- Чтение и запись буфферизировано.

Создать такой файл из терминала: `mkfifo fifoname`

Одноимённая библиотечная функция для создания программно – `mkfifo` (принимает путь куда положитть файл и права доступа, которые ему дать)

Пример общения между двумя процессами с помощью fifo-файла:

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

int main() {
    int res = mkfifo("myfifo", 0666);

    int fd = open("myfifo", O_WRONLY);
    char* str = "Hello!\n";
    while (true) {
        write(fd, str, strlen(str));
    }
}
```

Запускаете этот код, он пишет в фифо. В другом окне терминала (создать его – комбинация клавиш `Ctrl + Shift + T`) запускаете `cat myfifo` и процессы общаются друг с другом.

Что, если несколько процессов пишут в один и тот же fifo? Не было на лекции. Возможный ответ (похож на правду): С именованным каналом FIFO связан один буфер в ядре, вне зависимости от того, сколько процессов подключилось к этому каналу. Если писателей несколько, то порядок, в котором будут помещены их данные в буфер, не определён. Если данные, записываемые вызовом write(), длиннее длины буфера, атомарность записи не обеспечивается, и могут перемешаться куски данных, записываемые разными процессами.

Что, если несколько процессов читают один и тот же fifo? Тоже самое: Если несколько процессов читают один и тот же FIFO, данные из буфера будут передаваться в процессы-читатели в случайной очерёдности.

Пункт 31

Разделяемая память - третий оснвной вид `IPC`. Данный принцип основан на том, что процессы имеют общий сегмент памяти, запись в который из одного процесса видна в другом.

Для создания и управления разделяемой памятью есть два вида сисколлов:

1) Стандарт `System V` – считается устаревшим способом реализации shm. Сисколлы:

- `shmget(key_t key, size_t size, int shmflg)` – Получить `id` сегмента разделяемой памяти, если по такому ключу ещё нет сегментов, то указав флаг `IPC_CREAT` он создаст такой кусок
- `shmat(int shmid, const void _Nullable shmaddr, int shmflg)` – прицепиться к разделяемому сегменту памяти с таким `id`
- `shmdt(const void *shmaddr)` – отцепиться от этого сегмента памяти
- `shmctl(int shmid, int op, struct shmid_ds *buf)` – совершает различные операции над сегментом разделяемой памяти. Мы использовали лишь такие аргументы, чтобы удалить этот сегмент `shmctl(shmid, IPC_RMID, NULL)`;

Посмотреть о всех shm созданных этим способом можно командой `cat /proc/sysvipc/shm` или `ipcs -m`

2) Стандарт `POSIX` – более современный стандарт. Сисколлы:

- `shm_open(const char *name, int oflag, mode_t mode)` – прицепиться или создать к сегменту разделяемой памяти. Здесь он задаётся названием файла, который ОС поддерживает (виртуальный файл). Вернёт `fd` файла
- `mmap` – замапить этот файл к себе в виртуальное пространство
- `shm_unlink(const char *name)` – удалить ShM файл.

Файлы создаются в директории `/dev/shm`.

Также оба вида у работающего процесса можно глянуть в поддиректрии `/proc/<pid>/maps`

Примеры на обы вида есть в лекции по ShM

Пункт 32

Треды - это будто дочерние процессы (лучше говорить подпроцессы), у которых почти все данные общие с исходным. Подробнее: динамическая и статическая памяти, файловые дескрипторы, файловое пространство (например `cwd`) у тредов разделяемые, стек у каждого треда свой. Хотя эти аргументы можно задавать, что именно они будут разделять.

Пример использования thread. Оно запускает поток прямо в конструкторе объекта.

```
#include <thread>
#include <iostream>
#include <unistd.h>
void job() {
```

```
    std::cout << "Pid: " << getpid() << " tgid: " << gettid() << "\n";
    std::cout.flush();
    getchar();
}

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

Метод `join` – Блокирует выполнение текущего потока до тех пор, пока вызываемый тред не завершит выполнение.

Метод `detach` – Отсоединяет поток от текущего контекста. То есть main может закончиться раньше и тред станет независимым процессом

Что происходит, если main завершается до завершения потоков? (ну будет вызван ни join ни detach) Деструктор треда вызовет стандартную функцию `std::terminate`.

Пункт 33

`race condition` – состояние гонки, когда разные треды конкурируют за ресурсы процессора, исполняясь поочерёдно.

Пример `UB` – код ниже без мьютекса. Пусть у нас есть глобальный вектор и каждый тред в функции 100 раз вставляет в него элемент. Тогда неизвестно какое будет поведение этого кода, так как один и тот же участок памяти будет изменяться из разных тредов конкурентно. Но если поместить создание вектора внутрь функции, то всё будет ок, так как `malloc` поддерживает многопоточность.

Чтобы обезопаситься от таких случаев, есть методы синхронизации. Один из них – `mutex` – примитив. Имеет два метода `lock` и `unlock`. Область кода между этими двумя методами будет выполняться только одним процессом. `mutex` должен быть глобальным (логично).

Один минус – если забыли написать `unlock` (или он почему-то не вызвался), то любой другой поток, вызвавший `lock` у этого мьютекса зависнет навсегда. То есть возникнет `deadlock`. Для решения этой проблемы есть шаблонный класс `lock_guard<std::mutex(m)>` – аналогично умному указателю в конструкторе вызывает `lock`, в деструкторе – `unlock`.

Пример:

```cpp
#include <thread>
#include <iostream>
#include <vector>

std::vector<int> v;
std::mutex m;
void job() {
    m.lock();
    for (int i = 0; i < 10000; ++i) {
        v.push_back(i);
    }
    m.unlock();
}
int main() {
    std::thread t(job);
    job();
    t.join();
}
```

Пункт 34

Ну вообще можно глянуть `strace` на программу с `std::tread`

Глобально – используем сисколл `clone` (подробно о нём в man). Главное в нём – флаг `CLONE_THREAD`, который как раз и говорит о том, что мы создаём именно тред. Так же создаём отдельный стек, на котором будет исполняться тред. Передаём в `clone` указатель на КОНЕЦ стека, так как он растёт вниз. Чтобы вызвать функцию, используем статическую функцию обёртку, аргументом в неё передаём указатель на сам класс. Кастим и вызываем сохранённую функцию работы. Чтобы реализовать `join` делаем `waitpid` на сохранённый `pid` (его вернул `clone`). Потом выставляем `pid` в -1 и проверяем его в деструкотре. Если он не -1, то падаем. Реализацию смотри в лекции (там сегфолт, но идейно вроде верно).

С точки зрения Linux, треды — это процессы, которые разделяют одно адресное пространство (и другие ресурсы) с другими процессами, если указаны соответствующие флаги (`CLONE_VM` и другие).

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.

Notes: 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.

`tgid` – `id` тред-группы (совпадает с `pid` лидера тред-группы)

`tid` – уникальный `id` каждого треда

`pid` – идентификатор процесса. Общий для всех потоков

Узнать – сисколлы

- `getpid` – возвращает `tgid` (`pid` лидера), то что в htop обозначено за `TGID`
- `gettid` – Возвращает `tid` (то что в htop обозначено за `PID`). Вот такая странная логика

Послать сигнал отдельному треду – сисколл `tgkill`

Лекция 5

Здесь пусто

Смотрите конспект Лизы или конспект лекций Дениса =)