2 задача

Системные вызовы

В файлике **могут** быть ошибки, может немного поехала нумерация :)

1. Проведите следующие эксперименты:
   a. запустите программу hello world из предыдущей задачи под strace:

```c
#include <stdio.h>

int main() {
    printf("Hello, World!\n");
    return 0;
}
```

```
gcc -o hello hello.c
strace ./hello
```

```
execve("./hello", ["./hello"], 0x7fff07326310 /* 61 vars */) = 0
brk(NULL)                               = 0x61ecfbded000
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x77551843c000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (Нет такого файла или каталога)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=76735, ...}) = 0
mmap(NULL, 76735, PROT_READ, MAP_PRIVATE, 3, 0) = 0x775518429000
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\220\243\2\0\0\0\0\0"..., 832) =
832
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) =
784
fstat(3, {st_mode=S_IFREG|0755, st_size=2125328, ...}) = 0
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) =
784
mmap(NULL, 2170256, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x775518200000
mmap(0x775518228000, 1605632, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE,
3, 0x28000) = 0x775518228000
mmap(0x7755183b0000, 323584, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1b0000)
= 0x7755183b0000
mmap(0x7755183ff000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3,
0x1fe000) = 0x7755183ff000
mmap(0x775518405000, 52624, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS,
-1, 0) = 0x775518405000
close(3)                                = 0
mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x775518426000
arch_prctl(ARCH_SET_FS, 0x775518426740) = 0
set_tid_address(0x775518426a10)         = 21583
set_robust_list(0x775518426a20, 24)     = 0
```

```
rseq(0x775518427060, 0x20, 0, 0x53053053) = 0
mprotect(0x7755183ff000, 16384, PROT_READ) = 0
mprotect(0x61ece0056000, 4096, PROT_READ) = 0
mprotect(0x775518474000, 8192, PROT_READ) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY}) = 0
munmap(0x775518429000, 76735)           = 0
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0), ...}) = 0
getrandom("\xde\x01\xe3\xaf\xe3\xbc\xa3\x94", 8, GRND_NONBLOCK) = 8
brk(NULL)                               = 0x61ecfbded000
brk(0x61ecfbe0e000)                     = 0x61ecfbe0e000
write(1, "Hello world!\n", 13Hello world!
)           = 13
exit_group(0)                           = ?
+++ exited with 0 +++
```

Мы совсем не пугаемся! Это просто список системных вызовов, другими словами :

Это вывод программы `strace`, которая **отслеживает** системные вызовы, выполняемые процессом.
Подробнее про вызовы:

1. Запуск программы

```
execve("./hello", ["./hello"], 0x7fff07326310 /* 61 vars */) = 0
```

- **`execve`** — системный вызов, который запускает программу `./hello`.
- Первый аргумент — путь к исполняемому файлу.
- Второй аргумент — массив аргументов командной строки (в данном случае только `"./hello"`).
- Третий аргумент — окружение (61 переменная окружения).
- Возвращает `0`, если программа успешно запущена.

> **Переменная окружения** — это динамическая именованная переменная, которая содержит информацию, используемую операционной системой и другими программами. Она представляет собой пару «имя-значение», где имя — это идентификатор переменной (например, `PATH`, `TEMP`, `USERNAME`), а значение — это строка данных, связанная с этим именем.

В отличие от переменных, объявленных внутри программы, переменные окружения *доступны всем программам и процессам*, запущенным в данной сессии пользователя или системы. Они позволяют передавать информацию между процессами без явного программирования механизмов обмена данными.

Примеры использования:

`PATH`: Указывает операционной системе, в каких директориях искать исполняемые файлы (программы) при запуске команд в командной строке.

`TEMP` : Определяет временный каталог, где программы могут хранить временные файлы.
`USERNAME` : Содержит имя текущего пользователя.
`HOME` : Указывает на домашний каталог пользователя.
Переменные, настраиваемые приложениями: Многие программы используют переменные окружения для настройки своего поведения (например, путь к конфигурационному файлу, уровень логирования).

Как переменные окружения влияют на программы:
Программы могут считывать значения переменных окружения во время своего выполнения и использовать их для принятия решений или настройки своих параметров. Это позволяет сделать программы более гибкими и переносимыми, так как они могут адаптироваться к различным средам без необходимости перекомпиляции или изменения исходного кода.

---

2. Инициализация памяти

```
brk(NULL)                               = 0x61ecfbded000
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x77551843c000
```

- **`brk(NULL)`** — возвращает текущий адрес конца кучи (heap).
- **`mmap`** — выделяет память:
  - `NULL` — ядро само выбирает адрес.
  - `8192` — размер выделяемой памяти (8 КБ).
  - `PROT_READ|PROT_WRITE` — память доступна для чтения и записи.
  - `MAP_PRIVATE|MAP_ANONYMOUS` — память не связана с файлом и приватна для процесса.
  - `-1` — файловый дескриптор не используется.
  - `0` — смещение в файле (не используется).

> Подробней про **`brk`**
>
> ### 1. `brk(NULL)`
>
> Системный вызов `brk` используется для управления границей кучи (heap) процесса. Куча — это область памяти, которая используется для динамического выделения памяти (например, через `malloc` в C).
>
> - **`brk(NULL)`** возвращает текущий адрес конца кучи. В данном случае:
>
>   brk(NULL) = 0×61ecfbded000
>
>   Это означает, что текущая граница кучи находится по адресу `0x61ecfbded000`.
> - Если бы мы передали в `brk` ненулевой адрес, например, `brk(адрес)`, то это изменило бы границу кучи на указанный адрес. Это может быть использовано для увеличения или уменьшения размера кучи.

Подробней про `mmap`

2. `mmap`

Системный вызов `mmap` используется для выделения памяти в адресном пространстве процесса.
В данном случае:

mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0×77551843c000

Разберем аргументы:

- **`NULL`** : Ядро само выбирает адрес для выделения памяти. Если бы мы указали конкретный адрес, ядро попыталось бы выделить память по этому адресу (если это возможно).
- **`8192`** : Размер выделяемой памяти — 8192 байта (8 КБ).
- **`PROT_READ|PROT_WRITE`** : Устанавливает права доступа к памяти. В данном случае память доступна для чтения и записи. Возможные флаги:
  - `PROT_READ` — память доступна для чтения.
  - `PROT_WRITE` — память доступна для записи.
  - `PROT_EXEC` — память доступна для выполнения кода.
  - `PROT_NONE` — память недоступна.
- **`MAP_PRIVATE|MAP_ANONYMOUS`** : Указывает на тип отображения памяти:
  - `MAP_PRIVATE` — выделенная память будет приватной для процесса. Изменения в этой памяти не будут видны другим процессам.
  - `MAP_ANONYMOUS` — память не связана с файлом. Это означает, что выделенная память будет инициализирована нулями и не будет связана с каким-либо файлом на диске.
- **`-1`** : Файловый дескриптор. В данном случае он не используется, так как память не связана с файлом (используется `MAP_ANONYMOUS`).
- **`0`** : Смещение в файле. В данном случае не используется, так как память не связана с файлом.
- **`0x77551843c000`** : Адрес, по которому была выделена память.

Эти системные вызовы часто используются в низкоуровневом программировании для управления памятью, например, в реализации функций `malloc` и `free` в стандартной библиотеке C.

---

3. Загрузка динамических библиотек

```
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (Нет такого файла или каталога)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=76735, ...}) = 0
```

```
mmap(NULL, 76735, PROT_READ, MAP_PRIVATE, 3, 0) = 0x775518429000
close(3)                                = 0
```

- **`access`** — проверка наличия файла `/etc/ld.so.preload` (используется для предзагрузки библиотек). В данном случае файл отсутствует (`ENOENT`).
- **`openat`** — открытие файла `/etc/ld.so.cache` , который содержит кэш путей к динамическим библиотекам.
- **`fstat`** — получение информации о файле (размер, права доступа и т.д.).
- **`mmap`** — отображение файла в память для быстрого доступа.
- **`close`** — закрытие файлового дескриптора.

> Подробнее про каждую команду.

Эта строка проверяет существование файла `/etc/ld.so.preload` . Этот файл, если он присутствует, содержит список динамических библиотек, которые должны быть загружены перед любыми другими библиотеками. Это позволяет переопределить стандартные библиотеки или добавить специфичные для приложения. `R_OK` означает, что проверяется только право на чтение. `ENOENT` — это стандартный код ошибки, означающий, что файл не найден. В данном случае файл `/etc/ld.so.preload` отсутствует, поэтому загрузка предзагружаемых библиотек пропускается.

1. `openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3`

Здесь происходит открытие файла `/etc/ld.so.cache` в режиме только для чтения (`O_RDONLY`). `AT_FDCWD` указывает на текущий рабочий каталог как базовый для пути. `O_CLOEXEC` — флаг, который закрывает файловый дескриптор автоматически при создании нового процесса (fork). Это важная мера безопасности, предотвращающая наследование дескриптора дочерними процессами. Результат — файловый дескриптор 3.

1. `fstat(3, {st_mode=S_IFREG|0644, st_size=76735, ...}) = 0`

Системный вызов `fstat` получает информацию о файле, открытом с дескриптором 3 (`/etc/ld.so.cache`). `st_mode=S_IFREG|0644` указывает, что файл является обычным файлом (`S_IFREG`) с правами доступа 0644 (чтение и запись для владельца, чтение для группы и других). `st_size=76735` показывает размер файла в байтах (примерно 75 КБ).

1. `mmap(NULL, 76735, PROT_READ, MAP_PRIVATE, 3, 0) = 0x775518429000`

Это критически важный шаг. `mmap` отображает содержимое файла `/etc/ld.so.cache` в виртуальную память процесса. `NULL` указывает, что ядро должно выбрать адрес для отображения. `76735` — размер отображаемой области (соответствует размеру файла). `PROT_READ` разрешает только чтение. `MAP_PRIVATE` создает частную копию отображаемого файла — изменения в отображенной области не будут отражаться в исходном файле. `3` — файловый дескриптор. `0` — смещение в файле (начало файла). Результат — виртуальный адрес `0x775518429000` , по которому теперь можно быстро обращаться к содержимому кэша.

1. `close(3) = 0`

Файловый дескриптор 3, связанный с `/etc/ld.so.cache`, закрывается. Файл уже отображен в памяти, поэтому дескриптор больше не нужен.

В итоге: Этот фрагмент показывает, как система эффективно загружает информацию о расположенных в системе динамических библиотеках. Использование `ld.so.cache` позволяет избежать медленного поиска библиотек в файловой системе при каждом запуске программы. Кэш содержит информацию о местоположении библиотек, их зависимостях и прочих метаданных. Отображение кэша в память с помощью `mmap` обеспечивает быстрый доступ к этой информации, ускоряя процесс запуска программ. Отсутствие `/etc/ld.so.preload` просто означает, что нет библиотек, которые нужно загрузить до стандартных.

---

4. Загрузка стандартной библиотеки C (libc)

```
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\220\243\2\0\0\0\0\0"..., 832) =
832
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) =
784
fstat(3, {st_mode=S_IFREG|0755, st_size=2125328, ...}) = 0
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) =
784
mmap(NULL, 2170256, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x775518200000
mmap(0x775518228000, 1605632, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE,
3, 0x28000) = 0x775518228000
mmap(0x7755183b0000, 323584, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1b0000)
= 0x7755183b0000
mmap(0x7755183ff000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3,
0x1fe000) = 0x7755183ff000
mmap(0x775518405000, 52624, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS,
-1, 0) = 0x775518405000
close(3)                                = 0
```

- **`openat`** — открытие файла стандартной библиотеки C (`libc.so.6`).
- **`read`** и **`pread64`** — чтение заголовков ELF-файла (формат исполняемых файлов в Linux).
- **`mmap`** — отображение секций библиотеки в память:
  - `.text` (код) — с правами `PROT_READ|PROT_EXEC`.
  - `.data` и `.bss` — с правами `PROT_READ|PROT_WRITE`.
- **`close`** — закрытие файлового дескриптора.

подробней

1. **`openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3`**

Этот системный вызов открывает файл библиотеки `libc.so.6`.

- `AT_FDCWD` : Указывает, что путь к файлу задан относительно текущего рабочего каталога процесса.
- `/lib/x86_64-linux-gnu/libc.so.6` : Путь к файлу стандартной библиотеки C.
- `O_RDONLY|O_CLOEXEC` : Флаги открытия файла:
  - `O_RDONLY` — файл открывается только для чтения.
  - `O_CLOEXEC` — файловый дескриптор будет автоматически закрыт при вызове `execve` (например, при запуске другой программы).
- `= 3` : Файловый дескриптор, возвращаемый системой. В данном случае это `3` , так как дескрипторы `0` , `1` и `2` обычно заняты (stdin, stdout, stderr).

---

2. `read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\220\243\2\0\0\0\0\0"..., 832) = 832`

Этот вызов читает первые 832 байта из файла `libc.so.6` .

- `3` : Файловый дескриптор, возвращенный `openat` .
- `"\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\220\243\2\0\0\0\0\0"...` : Это начало ELF-заголовка файла. Первые 4 байта (`\177ELF`) — это "магическое число", указывающее, что это ELF-файл.
- `832` : Количество байт, которые нужно прочитать.
- `= 832` : Фактическое количество прочитанных байт.

ELF-заголовок содержит метаинформацию о файле, такую как тип файла (исполняемый, библиотека и т.д.), архитектура (x86_64), версия и т.д.

---

3. `pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784`

Этот вызов читает данные из файла, начиная с определенного смещения.

- `3` : Файловый дескриптор.
- `"\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"...` : Данные, прочитанные из файла. Это часть ELF-заголовка, содержащая информацию о секциях и сегментах.
- `784` : Количество байт, которые нужно прочитать.
- `64` : Смещение в файле, с которого начинается чтение.
- `= 784` : Фактическое количество прочитанных байт.

---

4. `fstat(3, {st_mode=S_IFREG|0755, st_size=2125328, ...}) = 0`

Этот вызов получает информацию о файле.

- `3` : Файловый дескриптор.
- `st_mode=S_IFREG|0755` : Указывает, что это обычный файл (`S_IFREG`) с правами доступа `0755` (владелец может читать, писать и выполнять, остальные — только читать и выполнять).
- `st_size=2125328` : Размер файла — 2125328 байт (около 2 МБ).
- `= 0` : Успешное выполнение.

---

5. Инициализация потоков и памяти

```
mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x775518426000
arch_prctl(ARCH_SET_FS, 0x775518426740) = 0
set_tid_address(0x775518426a10)         = 21583
set_robust_list(0x775518426a20, 24)     = 0
rseq(0x775518427060, 0x20, 0, 0x53053053) = 0
mprotect(0x7755183ff000, 16384, PROT_READ) = 0
mprotect(0x61ece0056000, 4096, PROT_READ) = 0
mprotect(0x775518474000, 8192, PROT_READ) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY}) = 0
munmap(0x775518429000, 76735)           = 0
```

- `mmap` — выделение памяти для TLS (Thread Local Storage).
- `arch_prctl` — настройка регистра FS для TLS.
- `set_tid_address` — установка адреса для идентификатора потока.
- `set_robust_list` — настройка списка robust futex (для синхронизации потоков).
- `rseq` — настройка restartable sequences (механизм для ускорения работы с потоками).
- `mprotect` — изменение прав доступа к памяти (например, запрет записи).
- `prlimit64` — получение лимитов ресурсов (например, размера стека).
- `munmap` — освобождение памяти.

Подробней

1. `mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x775518426000`

Этот вызов выделяет память для **TLS (Thread Local Storage)** — области памяти, уникальной для каждого потока.

- `NULL` : Ядро выбирает адрес для выделения памяти.
- `12288` : Размер выделяемой памяти — 12 КБ.
- `PROT_READ|PROT_WRITE` : Память доступна для чтения и записи.
- `MAP_PRIVATE|MAP_ANONYMOUS` : Память приватна для процесса и не связана с файлом.
- `-1` : Файловый дескриптор не используется.
- `0` : Смещение в файле не используется.
- `0x775518426000` : Адрес, по которому выделена память.

TLS используется для хранения данных, которые должны быть уникальными для каждого потока (например, локальные переменные `thread_local` в C++).

---

2. `arch_prctl(ARCH_SET_FS, 0x775518426740) = 0`

Этот вызов настраивает регистр **FS**, который используется для доступа к TLS.

- `ARCH_SET_FS` : Указывает, что регистр FS должен быть установлен на указанный адрес.
- `0x775518426740` : Адрес, который будет записан в регистр FS. Это база для TLS.
- `= 0` : Успешное выполнение.

Регистр **FS** используется в архитектуре x86_64 для доступа к TLS. Каждый поток имеет свой собственный регистр FS, что позволяет каждому потоку иметь уникальную область TLS.

---

3. `set_tid_address(0x775518426a10) = 21583`

Этот вызов устанавливает адрес, по которому будет храниться идентификатор потока (TID).

- `0x775518426a10` : Адрес, по которому будет храниться TID.
- `21583` : Идентификатор потока (TID), возвращаемый системой.

Этот механизм используется для уведомления потока о его завершении (например, при вызове `pthread_exit`).

---

4. `set_robust_list(0x775518426a20, 24) = 0`

Этот вызов настраивает **список robust futex** для потока.

- `0x775518426a20` : Адрес списка robust futex.
- `24` : Размер списка.
- `= 0` : Успешное выполнение.

Robust futex — это механизм, который позволяет потокам безопасно синхронизироваться даже в случае аварийного завершения одного из потоков. Если поток завершается, не освободив мьютекс, robust futex автоматически разблокирует его.

---

5. `rseq(0x775518427060, 0x20, 0, 0x53053053) = 0`

Этот вызов настраивает **restartable sequences (rseq)** — механизм для ускорения работы с потоками.

- `0x775518427060` : Адрес структуры rseq.

- `0x20` : Размер структуры.
- `0` : Флаги.
- `0x53053053` : Сигнатура для проверки целостности.
- `= 0` : Успешное выполнение.

Restartable sequences позволяют потокам выполнять короткие критические секции без блокировок, что повышает производительность в многопоточных приложениях.

---

6. `mprotect(0x7755183ff000, 16384, PROT_READ) = 0`

Этот вызов изменяет права доступа к области памяти.

- `0x7755183ff000` : Адрес начала области памяти.
- `16384` : Размер области (16 КБ).
- `PROT_READ` : Память доступна только для чтения.
- `= 0` : Успешное выполнение.

`mprotect` используется для изменения прав доступа к памяти. В данном случае запись в эту область памяти запрещена.

---

7. `mprotect(0x61ece0056000, 4096, PROT_READ) = 0`

Аналогично предыдущему вызову, но для другой области памяти.

- `0x61ece0056000` : Адрес начала области памяти.
- `4096` : Размер области (4 КБ).
- `PROT_READ` : Память доступна только для чтения.
- `= 0` : Успешное выполнение.

---

8. `mprotect(0x775518474000, 8192, PROT_READ) = 0`

Еще один вызов `mprotect` для изменения прав доступа к памяти.

- `0x775518474000` : Адрес начала области памяти.
- `8192` : Размер области (8 КБ).
- `PROT_READ` : Память доступна только для чтения.
- `= 0` : Успешное выполнение.

---

9. `prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY}) = 0`

Этот вызов получает текущие лимиты ресурсов для процесса.

- `0` : PID процесса (0 означает текущий процесс).
- `RLIMIT_STACK` : Лимит для размера стека.
- `NULL` : Новые лимиты не устанавливаются.
- `{rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY}` : Текущие лимиты:
  - `rlim_cur=8192*1024` : Текущий лимит стека — 8 МБ.
  - `rlim_max=RLIM64_INFINITY` : Максимальный лимит стека не ограничен.
- `= 0` : Успешное выполнение.

---

10. `munmap(0x775518429000, 76735) = 0`

Этот вызов освобождает ранее выделенную область памяти.

- `0x775518429000` : Адрес начала области памяти.
- `76735` : Размер области.
- `= 0` : Успешное выполнение.

`munmap` используется для освобождения памяти, выделенной с помощью `mmap` .

Эти вызовы обеспечивают корректную работу многопоточных приложений в Linux.

---

6. Вывод "Hello, World!"

```
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0), ...}) = 0
getrandom("\xde\x01\xe3\xaf\xe3\xbc\xa3\x94", 8, GRND_NONBLOCK) = 8
brk(NULL)                               = 0x61ecfbded000
brk(0x61ecfbe0e000)                     = 0x61ecfbe0e000
write(1, "Hello world!\n", 13Hello world!
)          = 13
```

- `fstat` — получение информации о файловом дескрипторе `1` (stdout).
- `getrandom` — получение случайных данных (используется для ASLR или других целей).
- `brk` — увеличение кучи.
- `write` — вывод строки `"Hello, World!\n"` на стандартный вывод (stdout).

> Подробней

1. `fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0), ...}) = 0`

`fstat(1, ...)`: Это системный вызов, который получает информацию о файловом дескрипторе. Аргумент `1` — это стандартный вывод (stdout). Функция `fstat` возвращает структуру `stat`, содержащую информацию о файле или устройстве, связанном с дескриптором.
`{st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0), ...}`: Это часть структуры `stat`, возвращаемой `fstat`.
`st_mode=S_IFCHR|0620`: `S_IFCHR` указывает, что это символьное устройство (character device), такое как терминал или консоль. `0620` — это права доступа к файлу (в данном случае устройству). Это октальное число, означающее:
`0600`: только владелец может читать и писать.
`0020`: группа может только читать.
`st_rdev=makedev(0x88, 0)`: `st_rdev` содержит информацию об устройстве. `makedev` — функция, которая создает номер устройства из двух чисел (major и minor номера). `0x88` и `0` — это конкретные номера устройства, которые зависят от системы. Они идентифицируют конкретный терминал или консоль.
`= 0`: Возвращаемое значение 0 указывает на успешное выполнение `fstat`.

2. `getrandom("\xde\x01\xe3\xaf\xe3\xbc\xa3\x94", 8, GRND_NONBLOCK) = 8`

`getrandom(...)`: Этот системный вызов генерирует криптографически безопасные случайные данные. Он используется для различных целей, таких как:
Address Space Layout Randomization (ASLR): Случайное размещение сегментов памяти программы в адресном пространстве для повышения безопасности. Это предотвращает атаки, эксплуатирующие известные адреса памяти.
Генерация ключей: Создание случайных ключей для шифрования.
Других задач, требующих случайности: Например, генерация случайных чисел в играх.
`"\xde\x01\xe3\xaf\xe3\xbc\xa3\x94"`: Это 8 байтов случайных данных, сгенерированных `getrandom`. Значения представлены в шестнадцатеричном формате.
`8`: Размер буфера (в байтах), куда записываются случайные данные.
`GRND_NONBLOCK`: Флаг, указывающий, что вызов не должен блокироваться, если генератор случайных чисел не готов. Если данных нет, функция вернет ошибку. В данном случае, 8 байт были сгенерированы успешно.
`= 8`: Возвращаемое значение 8 указывает, что было сгенерировано 8 байтов.

3. `brk(NULL) = 0x61ecfbded000`

`brk(NULL)`: Этот вызов возвращает текущий адрес конца кучи (heap). Куча — это область памяти, используемая программой для динамического выделения памяти. `NULL` в качестве аргумента указывает на запрос текущего значения.
`0x61ecfbded000`: Это адрес в памяти, указывающий на текущий конец кучи.

4. `brk(0x61ecfbe0e000) = 0x61ecfbe0e000`

`brk(0x61ecfbe0e000)`: Этот вызов изменяет размер кучи. Он увеличивает кучу до адреса `0x61ecfbe0e000`. Это необходимо, если программе требуется больше памяти.
`0x61ecfbe0e000`: Новый адрес конца кучи. Возвращаемое значение совпадает с аргументом, подтверждая успешное изменение размера кучи.

```
5. write(1, "Hello world!\n", 13) = 13
```

`write(1, ...)` : Это системный вызов, который записывает данные в файл или устройство.
`1` : Файловый дескриптор стандартного вывода (stdout).
`"Hello world!\n"` : Строка, которая должна быть выведена.
`13` : Длина строки (включая символ новой строки `\n`).
`= 13` : Возвращаемое значение 13 указывает, что 13 байтов успешно записаны в стандартный вывод, тем самым выведя на консоль "Hello world!".

В итоге, лог демонстрирует стандартный процесс запуска простой программы, которая выводит "Hello, World!". `fstat` используется для получения информации о выводе, `getrandom` — для генерации случайных данных (возможно, для ASLR), `brk` — для управления размером кучи, а `write` — для вывода строки на консоль. Все вызовы завершились успешно (с кодом возврата 0).

---

7. Завершение программы

```
exit_group(0)                           = ?
+++ exited with 0 +++
```

- **`exit_group`** — завершение всех потоков процесса с кодом возврата `0` .
- `+++ exited with 0 +++` — программа завершилась успешно.

Чем обусловлено столько вызовов?

большое количество системных вызовов в выводе `strace` для даже простой программы, такой как "Hello, world!", обусловлено необходимостью взаимодействия программы с ядром операционной системы для выполнения самых базовых операций. Это не просто "запуск на ядре процессора", а гораздо более комплексный процесс, включающий в себя:

Управление памятью: Выделение памяти для кода программы, данных, стека и кучи (heap) происходит через системные вызовы `mmap` и `brk` . `mmap` используется для отображения файлов (например, библиотек) и выделения анонимной памяти. `brk` увеличивает размер кучи. Ядро управляет виртуальной памятью, предоставляя программе иллюзию непрерывного адресного пространства, в то время как физическая память управляется ядром.

Загрузка динамических библиотек: Программа "Hello, world!" использует стандартную C-библиотеку (`libc.so.6`). Загрузка и связывание этой библиотеки – это сложный процесс, включающий поиск библиотеки, чтение её содержимого, создание отображений в память (`mmap`) и установку связей между кодом программы и функциями библиотеки. Это все выполняется через системные вызовы.

Обработка сигналов: Системные вызовы, такие как `set_tid_address` и `set_robust_list`, связаны с обработкой сигналов. ***Сигналы – это способ асинхронного взаимодействия между процессами и ядром.***

Случайность: `getrandom` используется для генерации случайных чисел. Это связано с ASLR (Address Space Layout Randomization) – техникой безопасности, которая случайным образом размещает адресное пространство программы в памяти, затрудняя атакам предсказуемость расположения кода и данных.

Защита памяти: `mprotect` и другие вызовы обеспечивают защиту памяти. Ядро обеспечивает механизмы, препятствующие доступу одного процесса к памяти другого.

Ввод/вывод: Даже простой вывод на консоль требует нескольких системных вызовов: `fstat` получает информацию о файловом дескрипторе стандартного вывода, а `write` – собственно, осуществляет запись данных.

Ограничение памяти стека и кучи (stack и heap) — это не непосредственная причина множества системных вызовов, а скорее контекст, в котором они происходят. Ядро отслеживает использование памяти программой и устанавливает ограничения, чтобы предотвратить переполнение стека или исчерпание доступной памяти. Системные вызовы типа `brk` и `mmap` взаимодействуют с этим управлением памятью.

В итоге, **большое количество системных вызовов отражает сложность взаимодействия программы с операционной системой на низком уровне**, даже для самых простых задач. Каждый системный вызов представляет собой запрос к ядру, требующий обработки и выполнения определённой операции. Это необходимо для обеспечения безопасности, эффективности и надёжности работы программы.

---

1. **Анализ системных вызовов:**
   - Обратите внимание на вызов `write(1, "Hello, World!\n", 13)`. Это системный вызов, который используется для вывода строки на стандартный вывод (stdout).
   - `write` принимает три аргумента:
     - `int fd` — файловый дескриптор (1 — это stdout).
     - `const void *buf` — указатель на данные для записи.
     - `size_t count` — количество байт для записи.
   - Возвращает количество записанных байт или -1 в случае ошибки.
2. **Использование `write` вместо `printf`:**
   Измените программу, чтобы использовать `write` напрямую:

```
#include <unistd.h>

int main() {
    const char *msg = "Hello, World!\n";
    write(1, msg, 13);
```

```
    return 0;
}
```

Скомпилируйте и запустите под `strace`:

```
strace ./hello
```

Убедитесь, что вызов `write` присутствует в выводе.

3. **Создание обертки над `write` с использованием `syscall`:**
   Используйте функцию `syscall` из библиотеки `libc`:

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

int main() {
    const char *msg = "Hello, World!\n";
    syscall(SYS_write, 1, msg, 13);
    return 0;
}
```

Скомпилируйте и проверьте вывод `strace`.

---

b. Запуск `wget` или `curl` под `strace`

Если вы так же не поняли что происходит, то поясняю , мы хотим посмотреть системные вызовы при загрузки сайта kernel.org (http://kernel.org/index.html) с помощью команды `wget`

4. **Запустите команду под `strace`:**

```
strace -o wget.log wget kernel.org
```

Или, если используете `curl`:

```
strace -o curl.log curl kernel.org
```

5. **Получите статистику системных вызовов:**
   Используйте команду `strace -c` для получения статистики:

```
strace -c wget kernel.org
```

Вы увидите таблицу с количеством вызовов, временем выполнения и ошибками.

?

`-o wget.log`: Этот флаг указывает `strace`, куда записывать результаты мониторинга. В данном случае, все системные вызовы, выполненные `wget`, будут записаны в файл `wget.log`.

Без этого флага вывод отображался бы в терминале.

`wget kernel.org`: Это команда, которая запускается под мониторингом `strace`. `wget` – это утилита для скачивания файлов из сети. В данном примере она скачивает главную страницу сайта `kernel.org`.

В итоге, команда делает следующее:

Запускает программу `wget` для скачивания `kernel.org`. `strace` перехватывает все системные вызовы, которые делает `wget` (например, `open`, `read`, `write`, `connect`, `recv`, `send`, `socket`, и многие другие), и записывает подробную информацию о них в файл `wget.log`. Этот лог-файл будет содержать временные метки, имена системных вызовов, переданные аргументы и возвращаемые значения. Анализ этого файла может помочь понять, как `wget` работает на низком уровне, и выявить потенциальные проблемы.

Что можно увидеть в `wget.log`:

Файл `wget.log` будет содержать множество строк, каждая из которых описывает один системный вызов. Вы увидите такие детали, как:

Имя системного вызова: Например, `open("http://kernel.org/index.html", O_RDONLY)` или `connect(3, {sa_family=AF_INET, sin_port=htons(80), ...}, 16)`.
Аргументы: Аргументы, переданные в системный вызов. Для `open` это имя файла, для `connect` – IP-адрес и порт.
Возвращаемое значение: Результат выполнения системного вызова. Успех или неудача, дескриптор файла, и т.д.
Время: Временная метка каждого вызова.

```
nekoie@nekoie:~/Рабочий стол/programm_on_cpp/OCi/2$    strace -o wget.log wget kernel.org
--2025-02-19 21:33:00--  http://kernel.org/
Распознаётся kernel.org (kernel.org)… 139.178.84.217, 2604:1380:4641:c500::1
Подключение к kernel.org (kernel.org)|139.178.84.217|:80... соединение установлено.
HTTP-запрос отправлен. Ожидание ответа… 301 Moved Permanently
Адрес: https://kernel.org/ [переход]
--2025-02-19 21:33:01--  https://kernel.org/
Подключение к kernel.org (kernel.org)|139.178.84.217|:443... соединение установлено.
HTTP-запрос отправлен. Ожидание ответа… 200 OK
Длина: 16834 (16K) [text/html]
Сохранение в: ‘index.html’

index.html                                  100%
[=========================================================================================>]
16,44K  --.-KB/s    за 0,001s

2025-02-19 21:33:03 (22,4 MB/s) - ‘index.html’ сохранён [16834/16834]



nekoie@nekoie:~/Рабочий стол/programm_on_cpp/OCi/2$
   strace -c wget kernel.org
```

```
URL transformed to HTTPS due to an HSTS policy
--2025-02-19 21:33:30--  https://kernel.org/
Распознаётся kernel.org (kernel.org)… 139.178.84.217, 2604:1380:4641:c500::1
Подключение к kernel.org (kernel.org)|139.178.84.217|:443... соединение установлено.
HTTP-запрос отправлен. Ожидание ответа… 200 OK
Длина: 16834 (16K) [text/html]
Сохранение в: ‘index.html.1’

index.html.1                                    100%
[=============================================================================================>]
16,44K  --.-KB/s    за 0,001s

2025-02-19 21:33:32 (25,3 MB/s) - ‘index.html.1’ сохранён [16834/16834]

% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 18,09    0,000329          10        31           write
 13,74    0,000250           2       122         7 read
  7,31    0,000133          66         2           getdents64
  7,31    0,000133           2        53        24 openat
  7,09    0,000129           3        35           close
  5,83    0,000106           4        23           brk
  5,17    0,000094           3        26           futex
  5,06    0,000092           5        18         8 newfstatat
  3,96    0,000072          72         1           ftruncate
  3,68    0,000067           3        19           clock_gettime
  3,68    0,000067           8         8           pselect6
  3,35    0,000061           2        30           ioctl
  3,02    0,000055           3        15           getpid
  2,97    0,000054           2        24           getpgrp
  2,58    0,000047           1        36           fstat
  1,92    0,000035           0        51           mmap
  1,26    0,000023           2         9           fcntl
  1,21    0,000022          22         1           utimensat
  0,88    0,000016          16         1           statfs
  0,55    0,000010           3         3           lseek
  0,55    0,000010           5         2           getrandom
  0,44    0,000008           8         1           flock
  0,22    0,000004           1         3           getuid
  0,11    0,000002           1         2         1 setsockopt
  0,00    0,000000           0         3           poll
  0,00    0,000000           0        12           mprotect
  0,00    0,000000           0         2           munmap
  0,00    0,000000           0         5           rt_sigaction
  0,00    0,000000           0         2           pread64
  0,00    0,000000           0         1         1 access
  0,00    0,000000           0         7           socket
  0,00    0,000000           0         6         3 connect
  0,00    0,000000           0         1           sendto
  0,00    0,000000           0         2           recvfrom
  0,00    0,000000           0         2           recvmsg
  0,00    0,000000           0         1           bind
  0,00    0,000000           0         2           getsockname
  0,00    0,000000           0         1           execve
  0,00    0,000000           0         1           getgroups
  0,00    0,000000           0         1           arch_prctl
  0,00    0,000000           0         1           set_tid_address
  0,00    0,000000           0         1           set_robust_list
```

```
  0,00    0,000000           0         1           prlimit64
  0,00    0,000000           0         1           sendmmsg
  0,00    0,000000           0         1           rseq
------ ----------- ----------- --------- --------- ----------------
100,00    0,001819           3       570        44 total
```

Подробней

Что происходит в выводе?

1. Работа `wget`:

- Программа `wget` подключается к сайту `kernel.org` по HTTP (порт 80), но получает ответ `301 Moved Permanently`, который указывает на то, что сайт доступен только по HTTPS.
- Затем `wget` подключается к `kernel.org` по HTTPS (порт 443) и скачивает файл `index.html`.

2. Статистика системных вызовов:

После завершения работы `wget` выводится таблица с информацией о системных вызовах. Разберем её:

- **`% time`**: Процент времени, затраченного на выполнение каждого типа системного вызова.
- **`seconds`**: Общее время, затраченное на выполнение всех вызовов данного типа.
- **`usecs/call`**: Среднее время выполнения одного вызова (в микросекундах).
- **`calls`**: Количество вызовов данного типа.
- **`errors`**: Количество ошибок, возникших при выполнении вызовов данного типа.
- **`syscall`**: Название системного вызова.

Основные системные вызовы:

1. **`write`**:
   - Используется для записи данных (например, в файл или на экран).
   - В данном случае `wget` записывает скачанные данные в файл `index.html`.
2. **`read`**:
   - Используется для чтения данных (например, из файла или сети).
   - `wget` читает данные, полученные от сервера.
3. **`openat`**:
   - Открывает файлы или директории.
   - `wget` использует этот вызов для открытия файлов (например, для сохранения скачанных данных).
4. **`close`**:

- Закрывает файловые дескрипторы.
- `wget` закрывает файлы после завершения работы с ними.

5. **`brk`**:
   - Управляет размером кучи (heap) процесса.
   - Используется для выделения памяти.
6. **`futex`**:
   - Используется для синхронизации потоков.
   - `wget` может использовать многопоточность для ускорения загрузки.
7. **`newfstatat`**:
   - Получает информацию о файле (например, размер).
   - `wget` проверяет свойства файлов.
8. **`mmap`**:
   - Отображает файлы или устройства в память.
   - Используется для работы с большими файлами.
9. **`socket`**, **`connect`**, **`recvfrom`**, **`sendto`**:
   - Используются для работы с сетью.
   - `wget` устанавливает соединение с сервером и обменивается данными.
10. **`execve`**:
    - Запускает новую программу.
    - В данном случае `wget` запускается как отдельный процесс.

Ошибки:

В таблице указаны ошибки (столбец `errors`). Например:

- **`openat`**: 24 ошибки. Это может быть связано с попытками открыть несуществующие файлы или директории.
- **`read`**: 7 ошибок. Возможно, это попытки чтения из закрытых файловых дескрипторов.
- **`setsockopt`**: 1 ошибка. Это может быть связано с настройкой параметров сокета.

Итог:

- **`strace`** показывает, как программа взаимодействует с ядром ОС через системные вызовы.
- В данном случае `wget` использует множество системных вызовов для работы с сетью, файлами и памятью.
- Статистика помогает понять, какие системные вызовы занимают больше всего времени и где возникают ошибки.

Разберитесь как устроена функция syscall(). Напишите код, который напечатает hello world без использования функции syscall().

1. Давайте устанавливать компилятор для ассемблера :)

```
sudo apt update
sudo apt install nasm
```

Сейчас мы напишем программу на ассмеблере, но ИСПОЛЬЗУЯ syscall !

2. Создание программы на ассемблере

Создайте файл с именем `hello.asm` и вставьте в него следующий код:

```
section .data
    msg db 'Hello, World!', 0xA  ; Сообщение для вывода, 0xA - это символ новой строки
    len equ $ - msg              ; Длина сообщения

section .text
    global _start

_start:
    ; Системный вызов write(1, msg, len)
    mov rax, 1                   ; Номер системного вызова write
    mov rdi, 1                   ; Файловый дескриптор (1 - стандартный вывод)
    mov rsi, msg                 ; Указатель на строку
    mov rdx, len                 ; Длина строки
    syscall                      ; Вызов ядра

    ; Системный вызов exit(0)
    mov rax, 60                  ; Номер системного вызова exit
    mov rdi, 0                   ; Код возврата
    syscall                      ; Вызов ядра
```

- `section .data`: Здесь мы определяем данные, которые будут использоваться в программе. В данном случае это строка "Hello, World!" и её длина.
- `section .text`: Здесь находится код программы.
- `_start`: Точка входа в программу.
- `mov rax, 1`: Загружаем номер системного вызова `write` в регистр `rax`.
- `mov rdi, 1`: Загружаем файловый дескриптор (1 - стандартный вывод) в регистр `rdi`.
- `mov rsi, msg`: Загружаем адрес строки в регистр `rsi`.
- `mov rdx, len`: Загружаем длину строки в регистр `rdx`.
- `syscall`: Выполняем системный вызов.
- `mov rax, 60`: Загружаем номер системного вызова `exit` в регистр `rax`.
- `mov rdi, 0`: Загружаем код возврата (0) в регистр `rdi`.
- `syscall`: Завершаем программу.

3. Сборка программы

Шаг 1: Ассемблирование

Скомпилируйте программу с помощью `nasm`. Для этого выполните команду:

```
nasm -f elf64 hello.asm -o hello.o
```

- `-f elf64`: Указывает формат выходного файла (в данном случае, 64-битный ELF-файл).
- `hello.asm`: Исходный файл на ассемблере.
- `-o hello.o`: Имя выходного объектного файла.

Шаг 2: Компоновка

Создайте исполняемый файл с помощью компоновщика `ld`:

```
ld hello.o -o hello
```

- `hello.o`: Объектный файл, полученный на предыдущем шаге.
- `-o hello`: Имя выходного исполняемого файла.

Шаг 3: Запуск программы

После успешной сборки вы получите исполняемый файл `hello`. Запустите его:

```
./hello
```

Вы увидите вывод:

Hello, World!

```
                                                    Умнички! <3
```

- В архитектуре x86-64 системные вызовы выполняются с использованием инструкции `syscall`.
- Согласно соглашениям, номер системного вызова должен быть помещен в регистр `rax`. Это требование архитектуры и операционной системы.
  - Почему важно использовать exit? - потому что процессор глупый ему нужно сказать остановится , иначе планировщик команд будет продолжать читать "команды" только уже из оперативной памяти вылезет segmentation fault, можете попробовать !

Что происходит под капотом?

1. **Ассемблирование**:

- `nasm` преобразует текстовый код на ассемблере в машинный код (объектный файл `hello.o`).

2. **Компоновка**:
   - `ld` связывает объектный файл с библиотеками (если они используются) и создает исполняемый файл.
3. **Запуск**:
   - Исполняемый файл загружается в память, и процессор начинает выполнять инструкции.

Разберитесь как работает системный вызов ptrace(2)

`ptrace(2)` — это мощный системный вызов, который позволяет одному процессу (называемому "трассировщиком") наблюдать и контролировать выполнение другого процесса (называемого "трассируемым"). С помощью `ptrace` можно:

- Читать и изменять память и регистры трассируемого процесса.
- Перехватывать системные вызовы и сигналы.
- Останавливать и возобновлять выполнение трассируемого процесса.

`ptrace` широко используется в отладчиках (например, `gdb`), а также в инструментах для анализа поведения программ, таких как `strace`.

Основные функции `ptrace`

Основные запросы (команды), которые можно передать в `ptrace`:

- `PTRACE_TRACEME`: Процесс сообщает, что он хочет быть отслеживаемым своим родительским процессом.
- `PTRACE_ATTACH`: Присоединиться к уже существующему процессу для его отслеживания.
- `PTRACE_DETACH`: Отсоединиться от процесса.
- `PTRACE_PEEKTEXT`, `PTRACE_POKETEXT`: Чтение и запись в память процесса.
- `PTRACE_GETREGS`, `PTRACE_SETREGS`: Чтение и запись регистров процесса.
- `PTRACE_SYSCALL`: Остановить процесс на входе и выходе из системного вызова.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ptrace.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/user.h>
#include <sys/reg.h>
#include <sys/syscall.h>

void trace_syscalls(pid_t child_pid) {
    int status;
```

```
    struct user_regs_struct regs;

    while (1) {
        // Ожидаем остановки дочернего процесса
        wait(&status);

        if (WIFEXITED(status)) {
            break;
        }

        // Получаем регистры дочернего процесса
        ptrace(PTRACE_GETREGS, child_pid, NULL, &regs);

        // Выводим номер системного вызова
        printf("Syscall %ld called\n", regs.orig_rax);

        // Продолжаем выполнение дочернего процесса до следующего системного вызова
        ptrace(PTRACE_SYSCALL, child_pid, NULL, NULL);
    }
}

int main() {
    pid_t child_pid;

    child_pid = fork();

    if (child_pid == 0) {
        // Дочерний процесс
        ptrace(PTRACE_TRACEME, 0, NULL, NULL);
        execl("/bin/ls", "ls", NULL);
    } else {
        // Родительский процесс
        trace_syscalls(child_pid);
    }

    return 0;
}
```

1. **fork()**: Создаем дочерний процесс. В дочернем процессе вызываем `ptrace(PTRACE_TRACEME)`, чтобы разрешить трассировку, и затем запускаем программу `/bin/ls` с помощью `execl()`.
2. **trace_syscalls()**: В родительском процессе мы ожидаем остановки дочернего процесса с помощью `wait()`. Если процесс завершился, выходим из цикла. Иначе, получаем регистры дочернего процесса с помощью `ptrace(PTRACE_GETREGS)` и выводим номер системного вызова (хранится в `regs.orig_rax`).
3. **PTRACE_SYSCALL**: После вывода информации о системном вызове, мы продолжаем выполнение дочернего процесса до следующего системного вызова с помощью `ptrace(PTRACE_SYSCALL)`.

Что в коде происходит??

`pid_t` — это тип данных, определенный в стандартной библиотеке C (`<sys/types.h>`). Он используется для хранения идентификаторов процессов (Process ID, PID). Каждый процесс в Linux имеет уникальный PID, который является целым числом.

```c
int main() {
    pid_t child_pid; // Получить PID текущего процесса

    child_pid = fork();
```

Функция `fork()` создает новый процесс (дочерний), который является почти точной копией родительского процесса. После вызова `fork()`:

- В родительском процессе `fork()` возвращает PID дочернего процесса.
- В дочернем процессе `fork()` возвращает `0`.

```c
child_pid = fork();
if (child_pid == 0) {
    // Дочерний процесс
} else {
    // Родительский процесс
}
```

`WIFEXITED` — что это?

`WIFEXITED` — это макрос, определенный в `<sys/wait.h>`. Он используется для проверки, завершился ли процесс нормально (например, с помощью вызова `exit()` или возврата из `main()`).

- Если процесс завершился нормально, `WIFEXITED(status)` возвращает `true`.
- Если процесс завершился аварийно (например, из-за сигнала), `WIFEXITED(status)` возвращает `false`.

```c
int status;
wait(&status);
if (WIFEXITED(status)) {
    printf("Процесс завершился нормально с кодом %d\n", WEXITSTATUS(status));
}
```

`ptrace` — это системный вызов, который позволяет родительскому процессу отслеживать и контролировать выполнение дочернего процесса.

- `PTRACE_GETREGS` — это команда, которая запрашивает у ядра регистры процессора дочернего процесса.
- `child_pid` — PID дочернего процесса.
- `NULL` — не используется в данном случае.
- `&regs` — указатель на структуру `struct user_regs_struct`, в которую будут записаны значения регистров.

```c
struct user_regs_struct {
    unsigned long orig_rax;  // Номер системного вызова
    unsigned long rax;       // Возвращаемое значение системного вызова
```

```
    // Другие регистры...
};
```

`execl` — это функция из семейства `exec`, которая заменяет текущий образ процесса новым образом (загружает новую программу в память и начинает её выполнение).

- Первый аргумент (`"/bin/ls"`) — это путь к исполняемому файлу.
- Второй аргумент (`"ls"`) — это имя программы, которое будет передано в `argv[0]`.
- Последний аргумент (`NULL`) обозначает конец списка аргументов.

```
execl("/bin/ls", "ls", "-l", NULL);  // Запуск "ls -l"
```

`PTRACE_SYSCALL` — это команда для `ptrace`, которая:

- Останавливает дочерний процесс на входе в системный вызов.
- После того как родительский процесс прочитает регистры (например, с помощью `PTRACE_GETREGS`), дочерний процесс продолжает выполнение.
- Когда дочерний процесс завершает системный вызов, он снова останавливается, и родительский процесс может прочитать результат.

```
ptrace(PTRACE_SYSCALL, child_pid, NULL, NULL);
```

ПодитожЫм :)

1. Родительский процесс создает дочерний процесс с помощью `fork()`.
2. В дочернем процессе:
   - Вызывается `ptrace(PTRACE_TRACEME)`, чтобы разрешить родительскому процессу отслеживать его.
   - Вызывается `execl("/bin/ls", "ls", NULL)`, чтобы заменить текущий процесс на `/bin/ls`.
3. В родительском процессе:
   - Вызывается `trace_syscalls(child_pid)`, чтобы отслеживать системные вызовы дочернего процесса.
   - С помощью `ptrace(PTRACE_SYSCALL, ...)` родительский процесс останавливает дочерний процесс на каждом системном вызове.
   - С помощью `ptrace(PTRACE_GETREGS, ...)` родительский процесс читает регистры дочернего процесса и выводит номер системного вызова.
4. Когда дочерний процесс завершается, родительский процесс выходит из цикла.

Что вывело?

nekoie@nekoie:~/Рабочий стол/programm_on_cpp/OCi/2$ ./ptrace
Syscall 59 called
Syscall 12 called
Syscall 12 called
Syscall 9 called

Syscall 9 called
Syscall 21 called
Syscall 21 called
Syscall 257 called
Syscall 257 called
Syscall 5 called
Syscall 5 called
Syscall 9 called
Syscall 9 called
Syscall 3 called
Syscall 3 called
Syscall 257 called
Syscall 257 called
Syscall 0 called
Syscall 0 called
Syscall 5 called
Syscall 5 called
Syscall 9 called
Syscall 9 called
Syscall 9 called
Syscall 9 called
Syscall 9 called
Syscall 9 called
Syscall 9 called
Syscall 9 called
Syscall 9 called
Syscall 9 called
Syscall 3 called
Syscall 3 called
Syscall 257 called
Syscall 257 called
Syscall 0 called
Syscall 0 called
Syscall 17 called
Syscall 17 called
Syscall 5 called
Syscall 5 called
Syscall 17 called
Syscall 17 called
Syscall 9 called
Syscall 9 called
Syscall 9 called
Syscall 9 called
Syscall 9 called
Syscall 9 called
Syscall 9 called
Syscall 9 called
Syscall 9 called

Syscall 9 called
Syscall 3 called
Syscall 3 called
Syscall 257 called
Syscall 257 called
Syscall 0 called
Syscall 0 called
Syscall 5 called
Syscall 5 called
Syscall 9 called
Syscall 9 called
Syscall 9 called
Syscall 9 called
Syscall 9 called
Syscall 9 called
Syscall 9 called
Syscall 9 called
Syscall 3 called
Syscall 3 called
Syscall 9 called
Syscall 9 called
Syscall 158 called
Syscall 158 called
Syscall 218 called
Syscall 218 called
Syscall 273 called
Syscall 273 called
Syscall 334 called
Syscall 334 called
Syscall 10 called
Syscall 10 called
Syscall 10 called
Syscall 10 called
Syscall 10 called
Syscall 10 called
Syscall 10 called
Syscall 10 called
Syscall 10 called
Syscall 10 called
Syscall 10 called
Syscall 302 called
Syscall 302 called
Syscall 11 called
Syscall 11 called
Syscall 137 called
Syscall 137 called
Syscall 137 called
Syscall 137 called
Syscall 318 called

```
Syscall 318 called
Syscall 12 called
Syscall 12 called
Syscall 12 called
Syscall 12 called
Syscall 257 called
Syscall 257 called
Syscall 5 called
Syscall 5 called
Syscall 0 called
Syscall 0 called
Syscall 0 called
Syscall 0 called
Syscall 3 called
Syscall 3 called
Syscall 21 called
Syscall 21 called
Syscall 257 called
Syscall 257 called
Syscall 5 called
Syscall 5 called
Syscall 9 called
Syscall 9 called
Syscall 3 called
Syscall 3 called
Syscall 16 called
Syscall 16 called
Syscall 16 called
Syscall 16 called
Syscall 257 called
Syscall 257 called
Syscall 5 called
Syscall 5 called
Syscall 217 called
Syscall 217 called
Syscall 217 called
Syscall 217 called
Syscall 3 called
Syscall 3 called
Syscall 5 called
Syscall 5 called
Syscall 1 called
hello hello.asm hello.c hello.o hello_syscall hello_syscall.c hello_without_sysconf
hello_without_sysconf.c hello_write hello_write.c ptrace ptrace.c
Syscall 1 called
Syscall 3 called
Syscall 3 called
Syscall 3 called
```

```
Syscall 3 called
Syscall 231 called
```

перечислены номера системных вызовов, которые были выполнены дочерним процессом (в данном случае, программой `/bin/ls`).

Разберем что происходит в выводе:

Основные системные вызовы в выводе

1. **Syscall 59**: `execve`
   Это системный вызов, который используется для запуска новой программы. В вашем случае он был вызван для запуска `/bin/ls`.
2. **Syscall 12**: `brk`
   Используется для изменения размера кучи процесса. Обычно вызывается при выделении памяти.
3. **Syscall 9**: `mmap`
   Используется для отображения файлов или устройств в память. Часто используется для выделения больших блоков памяти.
4. **Syscall 21**: `access`
   Проверяет, есть ли у процесса права доступа к файлу или директории.
5. **Syscall 257**: `openat`
   Открывает файл или директорию. Это более современная версия системного вызова `open`.
6. **Syscall 5**: `fstat`
   Получает информацию о файле (например, его размер, права доступа и т.д.).
7. **Syscall 3**: `read`
   Читает данные из файлового дескриптора (например, из файла или сокета).
8. **Syscall 0**: `read`
   Это также системный вызов `read`, но в некоторых системах он может быть вызван с другим контекстом.
9. **Syscall 1**: `write`
   Записывает данные в файловый дескриптор. В вашем случае он был использован для вывода содержимого директории на экран.
10. **Syscall 231**: `exit_group`
    Этот системный вызов завершает выполнение всей группы процессов. Он вызывается, когда программа завершает свою работу.

Что такое `exit_group` (Syscall 231)?

`exit_group` — это системный вызов, который завершает выполнение не только текущего процесса, но и всех потоков в его группе. Он используется для корректного завершения многопоточных программ. В отличие от `exit`, который завершает только текущий поток, `exit_group` гарантирует, что все потоки будут завершены.

Почему он появился в выводе?

Когда программа `/bin/ls` завершает свою работу, она вызывает `exit_group`, чтобы завершить все свои потоки (если они есть) и освободить ресурсы. Это стандартное поведение для многих программ, особенно тех, которые используют многопоточность.

В Linux системные вызовы имеют номера, которые могут отличаться в зависимости от архитектуры (x86, x86_64, ARM и т.д.). Для x86_64 таблицу системных вызовов можно найти в файле:

```
/usr/include/asm/unistd_64.h
```

Как меняется режим с пользовательского на ядерный

Когда программа выполняет системный вызов, процессор переключается из пользовательского режима в режим ядра. Это происходит следующим образом:

1. Программа загружает номер системного вызова в регистр `rax` и аргументы в другие регистры.
2. Программа выполняет инструкцию `syscall`.
3. Процессор переключается в режим ядра и начинает выполнение кода ядра, связанного с этим системным вызовом.
4. После завершения системного вызова процессор возвращается в пользовательский режим, и выполнение программы продолжается.

Как устроены прерывания и как они связаны со системными вызовами

Прерывания — это механизм, который позволяет процессору временно приостановить выполнение текущей программы и переключиться на выполнение обработчика прерываний. Прерывания могут быть вызваны аппаратными устройствами (например, клавиатурой) или программно (например, системными вызовами).

Как происходит прерывание:

1. Устройство (например, клавиатура) генерирует сигнал прерывания.
2. Процессор приостанавливает выполнение текущей программы и сохраняет её состояние (регистры, указатель инструкции и т.д.).
3. Процессор переходит к выполнению обработчика прерываний, который находится в ядре.
4. После завершения обработки прерывания процессор восстанавливает состояние программы и продолжает её выполнение.

Связь прерываний и системных вызовов:

Системные вызовы часто реализуются через программные прерывания. Например, в архитектуре x86 системные вызовы могут быть выполнены через инструкцию `int 0x80`, которая вызывает

прерывание с номером 0x80. Обработчик этого прерывания в ядре выполняет соответствующий системный вызов.

Что происходит при нажатии клавиши на клавиатуре

Когда вы нажимаете клавишу на клавиатуре:

1. Клавиатура генерирует сигнал прерывания.
2. Процессор приостанавливает выполнение текущей программы и переходит к выполнению обработчика прерывания клавиатуры.
3. Обработчик прерывания считывает код нажатой клавиши из буфера клавиатуры.
4. Обработчик прерывания передает код клавиши в соответствующую очередь в ядре.
5. Процессор возвращается к выполнению прерванной программы.

Номер прерывания

Номер прерывания — это уникальный идентификатор, который используется для определения типа прерывания. Например, в архитектуре x86:

- Прерывание 0x80 используется для системных вызовов.
- Прерывание 0x09 используется для обработки прерываний от клавиатуры.

как связаны kernel mode, syscall, прерывания????

Kernel mode, **системные вызовы (syscall)** и **прерывания (interrupts)** тесно связаны между собой, так как все они являются механизмами взаимодействия между пользовательскими программами и операционной системой (ядром ОС).

1. Kernel Mode (Режим ядра)

Это привилегированный режим работы процессора, в котором выполняется код операционной системы. В этом режиме код имеет полный доступ к аппаратным ресурсам (память, устройства ввода-вывода и т.д.). Пользовательские программы работают в **user mode** (пользовательском режиме), где доступ к аппаратным ресурсам ограничен.

---

2. Системные вызовы (Syscall)

Системные вызовы — это интерфейс, через который пользовательские программы запрашивают услуги у ядра ОС. Например:

- Открытие файла (`open()`)
- Чтение данных (`read()`)
- Создание нового процесса (`fork()`)

Когда программа выполняет системный вызов, происходит следующее:

1. Программа переключается из **user mode** в **kernel mode**.
2. Ядро ОС выполняет запрошенную операцию.
3. После завершения операции управление возвращается в **user mode**, и программа продолжает выполнение.

---

3. Прерывания (Interrupts)

Прерывания — это механизм, который позволяет процессору временно приостановить выполнение текущей задачи и переключиться на выполнение другого кода (например, обработчика прерывания). Прерывания могут быть:

- **Аппаратные** (например, от клавиатуры, таймера или сетевой карты).
- **Программные** (например, системные вызовы).

Как они связаны?

a) Системные вызовы используют прерывания

Когда программа выполняет системный вызов, она инициирует программное прерывание (например, с помощью инструкции `int 0x80` на x86 или `syscall` на современных процессорах). Это прерывание переключает процессор в **kernel mode** и передает управление ядру ОС.

Пример:

1. Программа вызывает системный вызов, например, `write()`.
2. Процессор переключается в **kernel mode** через прерывание.
3. Ядро ОС выполняет запрошенную операцию (например, запись данных в файл).
4. После завершения ядро возвращает управление программе, и процессор переключается обратно в **user mode**.

b) Прерывания переключают процессор в kernel mode

Любое прерывание (аппаратное или программное) вызывает переключение процессора в **kernel mode**. Это необходимо, чтобы ядро ОС могло безопасно обработать прерывание, не нарушая работу пользовательских программ.

Пример:

1. Устройство (например, клавиатура) генерирует аппаратное прерывание.
2. Процессор приостанавливает выполнение текущей программы и переключается в **kernel mode**
3. Ядро ОС обрабатывает прерывание (например, считывает данные с клавиатуры).
4. После обработки процессор возвращается в **user mode**, и программа продолжает выполнение.

c) Kernel mode обеспечивает безопасность

Переключение в **kernel mode** через прерывания или системные вызовы позволяет ядру ОС контролировать доступ к аппаратным ресурсам. Это предотвращает возможность повреждения системы пользовательскими программами.

Пример взаимодействия:

1. Пользовательская программа вызывает системный вызов, например, `read()`.
2. Программа инициирует программное прерывание (например, `int 0x80`).
3. Процессор переключается в **kernel mode**.
4. Ядро ОС обрабатывает системный вызов (например, читает данные из файла).
5. Ядро возвращает результат в программу и переключает процессор обратно в **user mode**.

Вопросы и ответы! :

1. Где распечатка Hello world(?) , рассказать всё про write
   `write` — это **системный вызов** в Unix-подобных операционных системах, который используется для записи данных в файл или файловый дескриптор (например, стандартный вывод — `stdout`).

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

- `fd` — файловый дескриптор (например, `1` для `stdout`).
- `buf` — указатель на данные, которые нужно записать.
- `count` — количество байт для записи.

---

2. Зачем нужны системные вызовы
   Системные вызовы — это интерфейс между пользовательскими программами и ядром операционной системы. Они нужны для выполнения задач, которые требуют привилегий ядра, таких как:

- Работа с файлами (`open`, `read`, `write`).
- Управление процессами (`fork`, `exec`, `kill`).
- Сетевые операции (`socket`, `connect`, `send`).
- Управление памятью (`mmap`, `brk`).

Почему нельзя обойтись без системных вызовов?

- Пользовательские программы работают в **пользовательском режиме**, где у них нет прямого доступа к аппаратным ресурсам (например, к диску или сети).
- Ядро работает в **привилегированном режиме** и может выполнять такие операции.
- Системные вызовы обеспечивают безопасность и изоляцию процессов.

---

3. Можно ли считать, что это просто динамические библиотеки, или у них есть своя особенность?
   Нет, системные вызовы — это не просто динамические библиотеки. У них есть свои особенности:

- **Режим выполнения**: Системные вызовы выполняются в привилегированном режиме ядра, а динамические библиотеки — в пользовательском режиме.
- **Механизм вызова**: Системные вызовы используют специальные инструкции процессора (например, `syscall` или `int 0x80` на x86), чтобы переключиться в режим ядра.
- **Назначение**: Системные вызовы предоставляют доступ к низкоуровневым функциям ОС, а динамические библиотеки — это просто наборы функций, которые могут быть загружены в память процесса.

---

4. Зачем для загрузки kernel. org нужны системные вызовы?
   Для загрузки сайта `kernel.org` (или любого другого сайта) используются системные вызовы, такие как:

- `socket` — создание сетевого сокета.
- `connect` — подключение к серверу.
- `send` / `recv` — отправка и получение данных.
  Эти вызовы необходимы, потому что:
- Сетевые операции требуют взаимодействия с сетевым оборудованием, доступ к которому возможен только через ядро.
- Ядро управляет сетевыми протоколами (TCP/IP) и обеспечивает безопасность.

---

5. Проблема, которая решается с помощью системных вызовов
   Основная проблема — **изоляция и безопасность**:

- Пользовательские программы не должны иметь прямого доступа к аппаратным ресурсам (например, к диску или сети), чтобы избежать повреждения данных или нарушения работы системы.
- Системные вызовы предоставляют контролируемый интерфейс для доступа к этим ресурсам.

---

6. Как меняется режим с пользовательского на ядерный?

1. **Режимы процессора**

Современные процессоры поддерживают несколько уровней привилегий (режимов), которые определяют, какие инструкции и ресурсы доступны программе. Обычно это:

- **Пользовательский режим (User Mode)**:
  - Ограниченный доступ к ресурсам.

    - Нельзя выполнять привилегированные инструкции (например, доступ к оборудованию или управление памятью).
    - Используется для выполнения пользовательских программ.
- **Режим ядра (Kernel Mode)**:
    - Полный доступ к ресурсам.
    - Можно выполнять любые инструкции, включая привилегированные.
    - Используется для выполнения кода операционной системы.

2. Как происходит переключение?

Переключение из пользовательского режима в режим ядра происходит через **системный вызов**. Давайте разберём этот процесс шаг за шагом.

Шаг 1: Программа инициирует системный вызов

Программа, работающая в пользовательском режиме, хочет выполнить операцию, которая требует привилегий ядра (например, запись данных на диск). Для этого она вызывает функцию, которая в конечном итоге приводит к выполнению системного вызова.

Пример:

```
write(1, "Hello, world!\n", 13);
```

Эта функция `write` из стандартной библиотеки C (`glibc`) внутри себя вызывает системный вызов.

Шаг 2: Выполнение инструкции системного вызова

Программа выполняет специальную инструкцию процессора, чтобы переключиться в режим ядра. На разных архитектурах это может быть:

- **x86 (32-bit)**: `int 0x80` (прерывание) или `sysenter`.
- **x86-64 (64-bit)**: `syscall`.
- **ARM**: `svc` (Supervisor Call).

Например, на x86-64:

```
mov rax, 1      ; Номер системного вызова (1 — write)
mov rdi, 1      ; Файловый дескриптор (stdout)
mov rsi, msg    ; Указатель на строку
mov rdx, 13     ; Длина строки
syscall         ; Вызов системного вызова
```

Шаг 3: Процессор переключается в режим ядра

Когда процессор выполняет инструкцию `syscall` (или аналогичную), происходит следующее:

1. **Сохранение состояния**:

- Процессор сохраняет текущее состояние программы:
  - Регистры (включая счётчик команд `RIP`).
  - Флаги (например, флаг разрешения прерываний).
  - Сегментные регистры (если используется сегментация).
- Это состояние сохраняется в специальной структуре, например, в **стеке ядра**.

2. **Переключение в режим ядра**:
   - Процессор переключается в режим ядра, что позволяет выполнять привилегированные инструкции.
3. **Переход к обработчику системного вызова**:
   - Процессор загружает адрес обработчика системных вызовов из специального регистра (например, `MSR_LSTAR` на x86-64).
   - Управление передаётся ядру операционной системы.

Шаг 4: Ядро выполняет запрошенную операцию

Ядро операционной системы получает управление и выполняет следующие действия:

1. **Определение системного вызова**:
   - Ядро смотрит на номер системного вызова (например, `1` для `write`), который был передан в регистре (например, `RAX` на x86-64).
2. **Выполнение операции**:
   - Ядро выполняет запрошенную операцию (например, запись данных на диск или вывод на экран).
   - Для этого оно может взаимодействовать с драйверами устройств или другими подсистемами ядра.
3. **Возврат результата**:
   - Результат операции (например, количество записанных байт) сохраняется в регистре (например, `RAX`).

Шаг 5: Возврат в пользовательский режим

После завершения операции ядро возвращает управление программе:

1. **Восстановление состояния**:
   - Процессор восстанавливает состояние программы из стека ядра:
     - Регистры.
     - Счётчик команд.
     - Флаги.
2. **Переключение в пользовательский режим**:
   - Процессор переключается обратно в пользовательский режим.
3. **Возврат управления программе**:
   - Управление передаётся программе, и она продолжает выполнение с точки, следующей за инструкцией `syscall`.

3. Роль таблицы системных вызовов

Ядро операционной системы использует **таблицу системных вызовов**, чтобы определить, какую операцию выполнить. Например:

- На Linux таблица системных вызовов определена в файле `arch/x86/entry/syscalls/syscall_64.tbl`.
- Каждому системному вызову соответствует номер и функция-обработчик.

Пример:

```
1   write   sys_write
2   read    sys_read
3   open    sys_open
```

Когда программа вызывает системный вызов, ядро ищет соответствующий обработчик в этой таблице.

4. **Роль прерываний**

Системные вызовы часто реализуются через **программные прерывания**. Например, на x86 (32-bit) используется прерывание `int 0x80`. Как это работает:

1. Программа вызывает `int 0x80`.
2. Процессор переключается в режим ядра и вызывает обработчик прерывания.
3. Обработчик прерывания определяет, что это системный вызов, и передаёт управление соответствующему обработчику системного вызова.

---

7. Как устроены прерывания и как они связаны со системными вызовами(как происходит это технически) - оберточка, связанная с прерываниями

1. **Что такое прерывания?**

Прерывания — это механизм, который позволяет процессору временно приостановить выполнение текущей задачи и переключиться на обработку важного события. Прерывания могут быть:

- **Аппаратными** (например, от клавиатуры, таймера или сетевой карты).
- **Программными** (например, системные вызовы через `int 0x80` или `syscall`).

2. **Как процессор обрабатывает прерывания?**

Шаг 1: Инициализация прерываний

При загрузке операционной системы ядро настраивает **таблицу прерываний** (Interrupt Descriptor Table, IDT). Эта таблица содержит адреса обработчиков прерываний для каждого типа прерывания.

- На x86 таблица прерываний называется **IDT**.
- Каждая запись в IDT содержит:
  - Адрес обработчика прерывания.
  - Уровень привилегий (DPL).
  - Тип шлюза (например, шлюз прерывания или шлюз ловушки).

Шаг 2: Процессор получает прерывание

Когда происходит прерывание (аппаратное или программное), процессор выполняет следующие действия:

1. **Определение типа прерывания**:
   - Для аппаратных прерываний контроллер прерываний (например, APIC или PIC) отправляет процессору номер прерывания.
   - Для программных прерываний номер прерывания указывается в инструкции (например, `int 0x80`).
2. **Сохранение состояния**:
   - Процессор сохраняет текущее состояние программы:
     - Счётчик команд (`RIP` или `EIP`).
     - Регистры флагов (`RFLAGS` или `EFLAGS`).
     - Сегментные регистры (если используется сегментация).
   - Это состояние сохраняется в **стеке ядра**.
3. **Переключение в режим ядра**:
   - Процессор переключается в режим ядра, чтобы выполнить обработчик прерывания.
4. **Поиск обработчика в IDT**:
   - Процессор использует номер прерывания как индекс в таблице IDT.
   - Из IDT извлекается адрес обработчика прерывания.
5. **Переход к обработчику**:
   - Процессор передаёт управление обработчику прерывания.

---

3. Обработка прерывания

Обработчик прерывания — это часть кода ядра, которая выполняет действия, необходимые для обработки прерывания. Например:

- Для аппаратного прерывания от клавиатуры обработчик читает скан-код и передаёт его драйверу клавиатуры.
- Для программного прерывания (системного вызова) обработчик выполняет запрошенную операцию (например, запись данных на диск).

Пример обработчика прерывания на ассемблере:

```
; Пример обработчика прерывания для int 0x80
interrupt_handler:
    pusha           ; Сохраняем все регистры
    ; Выполняем необходимые действия
```

```
    ; ...
    popa            ; Восстанавливаем регистры
    iret            ; Возврат из прерывания
```

---

4. Связь прерываний и системных вызовов

Системные вызовы часто реализуются через **программные прерывания**. Например, на x86 (32-bit) используется прерывание `int 0x80`. Как это работает:

1. **Программа вызывает системный вызов**:
   - Программа выполняет инструкцию `int 0x80` (или `syscall` на x86-64).
   - Номер системного вызова передаётся в регистре (например, `EAX`).
2. **Процессор обрабатывает прерывание**:
   - Процессор переключается в режим ядра и вызывает обработчик прерывания.
   - Обработчик прерывания определяет, что это системный вызов, и передаёт управление соответствующему обработчику системного вызова.
3. **Обработка системного вызова**:
   - Ядро выполняет запрошенную операцию (например, запись данных на диск).
   - Результат операции возвращается в регистре (например, `EAX`).
4. **Возврат в пользовательский режим**:
   - Процессор восстанавливает состояние программы и возвращает управление.

---

6. Роль контроллера прерываний

Для аппаратных прерываний используется **контроллер прерываний** (например, PIC или APIC). Его задачи:

- Управлять несколькими источниками прерываний (например, клавиатура, таймер, сеть).
- Отправлять процессору номер прерывания.
- Обеспечивать приоритет прерываний.

---

7. Возврат из прерывания

После завершения обработки прерывания процессор выполняет инструкцию `iret` (Interrupt Return), которая:

1. Восстанавливает состояние программы из стека ядра.
2. Переключает процессор обратно в пользовательский режим.
3. Возвращает управление программе.