**ОБЩАЯ ТЕОРИЯ**

**Системные вызовы (system calls)** — это механизм взаимодействия пользовательских программ с ядром операционной системы. Они позволяют программе выполнять **привилегированные операции**, такие как доступ к файловой системе, управление процессами, взаимодействие с сетью и памятью.

**Привилегированные операции** — это команды или действия, которые могут выполняться только в режиме ядра (kernel mode) и требуют повышенного уровня доступа. Эти операции критически важны для работы операционной системы и могут повлиять на безопасность и стабильность системы.

Когда программа хочет, например, открыть файл (open), прочитать данные (read) или создать новый процесс (fork), она не может делать это напрямую, так как работает в пользовательском режиме (user mode). Вместо этого она вызывает специальную функцию API (например, open() из стандартной библиотеки), которая в конечном итоге приводит к системному вызову.

Хотя системные вызовы лежат в памяти, их точные **адреса скрыты** от обычных программ. Это сделано по нескольким причинам:

1. Безопасность

Если бы программы знали адреса функций ядра, они могли бы пытаться вызывать их напрямую, обходя механизмы защиты.

Это открывало бы возможность атак, например, перезаписи таблицы системных вызовов или использования эксплойтов.

2. Абстракция и совместимость

Ядро ОС скрывает детали реализации, предоставляя интерфейс в виде номеров системных вызовов.

Адреса могут изменяться между версиями ОС, но интерфейс системных вызовов остаётся стабильным.

Вместо того чтобы программы знали адреса функций ядра, используется **таблица системных вызовов (System Call Table)**, содержащая соответствие номеров вызовов их обработчикам.

**Таблица системных вызовов (System Call Table)** — это структура данных в ядре операционной системы, содержащая список всех системных вызовов и их соответствующих обработчиков (функций ядра).

Когда программа делает системный вызов, процессор не выполняет код пользовательской программы напрямую, а обращается к таблице системных вызовов, чтобы найти нужную функцию в ядре и передать ей управление.

Таблица представляет собой массив указателей на функции ядра. Каждая строка содержит:

1) Номер системного вызова (идентификатор).

2) Адрес обработчика в ядре (указатель на функцию).

3) Количество аргументов (в некоторых реализациях).

1. Пользовательская программа вызывает API-функцию. Например, при read(fd, buf, size) фактически вызывается sys\_read.

2. Передача номера вызова в регистр. В x86-64 номер системного вызова помещается в регистр rax.

Аргументы передаются через rdi, rsi, rdx и другие регистры.

3. Выполняется инструкция syscall (или int 0x80 на старых x86). Процессор переключается в режим ядра (kernel mode). Обращается к таблице системных вызовов, используя номер вызова как индекс.

4. Обработчик системного вызова выполняется в ядре. Ядро выполняет соответствующую функцию (sys\_read). Производится работа с памятью, файлами, сетью и т. д.

5. Возврат результата. Обработчик завершает выполнение, возвращая значение в rax. Управление возвращается в пользовательский режим (user mode).

Пользовательская программа не вызывает функцию ядра напрямую, а делает вызов через **прерывание** или специальную инструкцию (например, syscall в x86-64).

**Прерывание (interrupt)** — это механизм, который позволяет процессору временно прерывать выполнение текущей задачи, чтобы обработать важное событие (например, ввод с клавиатуры, доступ к диску, сетевые пакеты и т. д.).

Типы прерываний:

1. Аппаратные (hardware interrupts)

Инициируются устройствами (например, клавиатурой, таймером, жёстким диском).

Пример: нажатие клавиши → генерируется прерывание → обработчик в ядре считывает символ.

2. Программные (software interrupts)

Генерируются программами, например, при выполнении системного вызова.

Например, на архитектуре x86 системные вызовы раньше вызывались через int 0x80, а теперь через syscall (более быстрый механизм).

**Как происходит системный вызов?**

1. Программа вызывает функцию из стандартной библиотеки

Например, read(fd, buf, size), которая фактически является оболочкой вокруг системного вызова sys\_read.

2. Передача управления ядру

Программа загружает номер системного вызова в регистр (eax в x86, rax в x86-64).

Аргументы передаются через другие регистры (rdi, rsi, rdx и т. д.).

3. Вызывается syscall (x86-64) или int 0x80 (x86).

4. Обработка в ядре

Процессор переключается в привилегированный режим (kernel mode).

5. Таблица системных вызовов используется для нахождения нужной функции (например, sys\_read).

Выполняется код ядра.

6. Возврат результата

Возвращаемое значение записывается в регистр rax (x86-64).

7. Управление передаётся обратно в пользовательский режим.

**Как происходит системный вызов?**

1. Программа вызывает функцию из стандартной библиотеки

Например, read(fd, buf, size), которая фактически является оболочкой вокруг системного вызова sys\_read.

2. Передача управления ядру

Программа загружает номер системного вызова в регистр (eax в x86, rax в x86-64).

Аргументы передаются через другие регистры (rdi, rsi, rdx и т. д.).

3. Вызывается syscall (x86-64) или int 0x80 (x86).

4. Обработка в ядре

Процессор переключается в привилегированный режим (kernel mode).

5. Таблица системных вызовов используется для нахождения нужной функции (например, sys\_read).

Выполняется код ядра.

6. Возврат результата

Возвращаемое значение записывается в регистр rax (x86-64).

7. Управление передаётся обратно в пользовательский режим. **ЗАДАНИЕ**

1. Проведите следующие эксперименты:

a. запустите программу hello world из предыдущей задачи под strace:

#include <stdio.h>

void PrintHello(void) {

printf("Hello, World!\n");

}

int main() {

PrintHello();

return 0;

}

*parallels@ubuntu-linux-2404:~/Desktop/OS$ strace ./hello*

*execve("./hello", ["./hello"], 0xffffe6eb5dd0 /\* 48 vars \*/) = 0*

*brk(NULL) = 0xcaa487345000*

*mmap(NULL, 8192, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0xff35d51a7000*

*faccessat(AT\_FDCWD, "/etc/ld.so.preload", R\_OK) = -1 ENOENT (No such file or directory)*

*openat(AT\_FDCWD, "/etc/ld.so.cache", O\_RDONLY|O\_CLOEXEC) = 3*

*fstat(3, {st\_mode=S\_IFREG|0644, st\_size=66427, ...}) = 0*

*mmap(NULL, 66427, PROT\_READ, MAP\_PRIVATE, 3, 0) = 0xff35d5196000*

*close(3) = 0*

*openat(AT\_FDCWD, "/lib/aarch64-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\267\0\1\0\0\0\360\206\2\0\0\0\0\0"..., 832) = 832*

*fstat(3, {st\_mode=S\_IFREG|0755, st\_size=1722920, ...}) = 0*

*mmap(NULL, 1892240, PROT\_NONE, MAP\_PRIVATE|MAP\_ANONYMOUS|MAP\_DENYWRITE, -1, 0) = 0xff35d4fa0000*

*mmap(0xff35d4fa0000, 1826704, PROT\_READ|PROT\_EXEC, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0) = 0xff35d4fa0000*

*munmap(0xff35d515e000, 65424) = 0*

*mprotect(0xff35d513a000, 77824, PROT\_NONE) = 0*

*mmap(0xff35d514d000, 20480, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x19d000) = 0xff35d514d000*

*mmap(0xff35d5152000, 49040, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_FIXED|MAP\_ANONYMOUS, -1, 0) = 0xff35d5152000*

*close(3) = 0*

*set\_tid\_address(0xff35d51a7f90) = 8740*

*set\_robust\_list(0xff35d51a7fa0, 24) = 0*

*rseq(0xff35d51a85e0, 0x20, 0, 0xd428bc00) = 0*

*mprotect(0xff35d514d000, 12288, PROT\_READ) = 0*

*mprotect(0xcaa4502bf000, 4096, PROT\_READ) = 0*

*mprotect(0xff35d51ac000, 8192, PROT\_READ) = 0*

*prlimit64(0, RLIMIT\_STACK, NULL, {rlim\_cur=8192\*1024, rlim\_max=RLIM64\_INFINITY}) = 0*

*munmap(0xff35d5196000, 66427) = 0*

*fstat(1, {st\_mode=S\_IFCHR|0620, st\_rdev=makedev(0x88, 0), ...}) = 0*

*getrandom("\x6d\x3a\xba\x21\xb0\xc3\x96\x89", 8, GRND\_NONBLOCK) = 8*

*brk(NULL) = 0xcaa487345000*

*brk(0xcaa487366000) = 0xcaa487366000*

*write(1, "Hello, World!\n", 14Hello, World!*

*) = 14*

*exit\_group(0) = ?*

*+++ exited with 0 +++*

1. *execve("./hello", ["./hello"], 0xffffe6eb5dd0 /\* 48 vars \*/) = 0*

Запускает новый процесс, заменяя код текущего процесса strace на исполняемый файл ./hello.

Синтаксис:

*int execve(const char \*filename, char \*const argv[], char \*const envp[]);*

filename = "./hello" → исполняемый файл.

argv = ["./hello"] → аргументы командной строки (массив char \*).

envp = 0xffffe6eb5dd0 → указатель на 48 переменных окружения (например, PATH, LD\_LIBRARY\_PATH и др.).

Что происходит внутри?

- Очищается память старого процесса.

- Загружается новый ELF-файл (./hello).

- Переносится управление в ./hello (начало выполнения main()).

2. *brk(NULL) = 0xcaa487345000*

Запрашивает текущую границу кучи (heap).

NULL → просто возвращает текущий адрес конца кучи (не меняет её размер).

3. *mmap(NULL, 8192, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0xff35d51a7000*

Выделяет 8 KB анонимной памяти (без файла).

**Анонимная память** — это область памяти, не связанная ни с каким файлом. Она создаётся в адресном пространстве процесса с помощью mmap() и используется, например, для стека, кучи, буферов, временных данных.

4. *faccessat(AT\_FDCWD, "/etc/ld.so.preload", R\_OK) = -1 ENOENT (No such file or directory)*

faccessat() проверяет доступность файла по указанному пути.

Синтаксис:

*int faccessat(int dirfd, const char \*pathname, int mode, int flags);*

dirfd = AT\_FDCWD → процесс проверяет файл относительно текущей директории.

pathname = "/etc/ld.so.preload" → проверяем существует ли этот файл.

mode = R\_OK → проверяем можно ли его прочитать.

flags (отсутствует) → используются стандартные настройки.

Почему файла нет?

Функция возвращает -1 ENOENT (No such file or directory), что означает:

Файл /etc/ld.so.preload не существует.

Этот файл не обязателен. Он используется для форсированной подгрузки библиотек перед остальными.

Обычно динамический загрузчик (ld-linux.so) подгружает библиотеки в стандартном порядке:

1) Ищет в LD\_LIBRARY\_PATH (если переменная окружения задана).

2) Ищет в кэше /etc/ld.so.cache (список известных библиотек).

3) Ищет в стандартных системных каталогах (/lib, /usr/lib и т. д.).

Но иногда нужно насильно загрузить конкретную библиотеку перед остальными. Для этого есть механизм форсированной подгрузки библиотек.

Если его нет → система просто пропускает этот шаг.

**/etc/ld.so.preload** - Список библиотек, которые подгружаются принудительно перед всеми другими. Может использоваться для принудительного подмены системных библиотек (например, для профилирования).

5. *openat(AT\_FDCWD, "/etc/ld.so.cache", O\_RDONLY|O\_CLOEXEC) = 3*

openat() открывает файл.

Синтаксис:

*int openat(int dirfd, const char \*pathname, int flags, mode\_t mode);*

dirfd = AT\_FDCWD → работать с файлом относительно текущей директории.

pathname = "/etc/ld.so.cache" → открыть файл кэша динамического линковщика.

flags = O\_RDONLY | O\_CLOEXEC:

O\_RDONLY → открыть файл только для чтения.

O\_CLOEXEC → если процесс создаст новый (fork() + execve()), файл закроется автоматически.

Функция возвращает 3, что значит, что файл успешно открыт, и 3 — это файловый дескриптор.

**Файловый дескриптор** (FD) — это просто целое число, которое ОС использует для отслеживания открытых файлов и других ресурсов (сокетов, каналов, терминалов и т. д.).

**/etc/ld.so.cache** - Бинарный кэш динамического линковщика (ld-linux), в котором хранятся пути к библиотекам. Позволяет системе быстро находить нужные библиотеки, без необходимости искать их по директориям.

6. *fstat(3, {st\_mode=S\_IFREG|0644, st\_size=66427, ...}) = 0*

Вызывает fstat() для файлового дескриптора 3, чтобы получить информацию о файле. Перед маппингом файла в память (mmap) нужно узнать его размер.

Синтаксис:

*fstat(fd, struct stat \*buf);*

3 — файловый дескриптор (файл /etc/ld.so.cache, открыт ранее через openat).

{st\_mode=S\_IFREG|0644, st\_size=66427, ...} — часть структуры stat:

S\_IFREG — обычный файл.

0644 — права (rw-r--r-- → владелец может писать и читать, остальные только читать).

st\_size=66427 — размер файла в байтах (≈ 64 KB).

= 0 — успешное выполнение.

7. *mmap(NULL, 66427, PROT\_READ, MAP\_PRIVATE, 3, 0) = 0xff35d5196000*

Загружает файл (/etc/ld.so.cache) в память. Загрузчик ld.so кэширует пути к библиотекам, чтобы ускорить поиск.

Синтаксис:

*mmap(addr, length, prot, flags, fd, offset);*

NULL — ОС сама выберет адрес в памяти.

66427 — размер памяти (файл целиком).

PROT\_READ — доступ только на чтение.

MAP\_PRIVATE — частное отображение (изменения не отразятся в файле).

3 — файловый дескриптор (/etc/ld.so.cache).

0 — смещение (загрузить с начала).

= 0xff35d5196000 — успешное маппирование, файл загружен по этому адресу.

8. *close(3) = 0*

Закрывает файловый дескриптор 3 (/etc/ld.so.cache). Файл уже загружен в память (mmap), его больше не нужно держать открытым.

9. *openat(AT\_FDCWD, "/lib/aarch64-linux-gnu/libc.so.6", O\_RDONLY|O\_CLOEXEC) = 3*

Открывает библиотеку libc.so.6.

AT\_FDCWD — искать относительно текущего каталога (/).

"/lib/aarch64-linux-gnu/libc.so.6" — путь к библиотеке.

O\_RDONLY — открыть только для чтения.

O\_CLOEXEC — закрыть при execve() (защита от утечек FD).

= 3 — успешно открыт, дескриптор 3.

10. *read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0\267\0\1\0\0\0\360\206\2\0\0\0\0\0"..., 832) = 832*

Читает первые 832 байта libc.so.6. Загрузчик проверяет, что это ELF-файл, и читает заголовок.

3 — дескриптор (libc.so.6).

"\177ELF\2\1\1..." — содержимое файла (начинается с \x7FELF, магическое число ELF).

832 — сколько байт прочитано.

11. *fstat(3, {st\_mode=S\_IFREG|0755, st\_size=1722920, ...}) = 0*

Получает информацию о файле с дескриптором 3 (в данном случае libc.so.6). Программе нужно знать размер libc.so.6, прежде чем загружать его в память.

3 — файловый дескриптор (libc.so.6, открыт ранее).

{st\_mode=S\_IFREG|0755, st\_size=1722920, ...}:

S\_IFREG — это обычный файл (не директория или устройство).

0755 — права доступа (rwxr-xr-x, исполняемый для всех).

st\_size=1722920 — размер файла (≈ 1.72 MB).

= 0 — успешно выполнено.

12. *mmap(NULL, 1892240, PROT\_NONE, MAP\_PRIVATE|MAP\_ANONYMOUS|MAP\_DENYWRITE, -1, 0) = 0xff35d4fa0000*

Выделяет анонимную область памяти размером 1.89 MB, но пока без доступа.

NULL — ОС сама выберет адрес (0xff35d4fa0000).

1892240 — размер (≈1.89 MB).

PROT\_NONE — никаких прав (пока нельзя читать/писать/исполнять).

MAP\_PRIVATE|MAP\_ANONYMOUS|MAP\_DENYWRITE:

MAP\_ANONYMOUS — без привязки к файлу (чистая память).

MAP\_DENYWRITE — запрет изменять оригинальный файл.

-1, 0 — т.к. MAP\_ANONYMOUS, файловый дескриптор не нужен.

13. *mmap(0xff35d4fa0000, 1826704, PROT\_READ|PROT\_EXEC, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0) = 0xff35d4fa0000*

Загружает код библиотеки libc.so.6 в память.

0xff35d4fa0000 — уже выделенный адрес.

1826704 — размер (≈1.82 MB).

PROT\_READ|PROT\_EXEC — доступ только на чтение и выполнение (код).

MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE:

MAP\_FIXED — загрузить точно в этот адрес.

MAP\_PRIVATE — изменения не попадут в файл.

3, 0 — fd=3 (libc.so.6), offset=0 (загрузка с начала).

14. *munmap(0xff35d515e000, 65424) = 0*

Освобождает ненужную память.

0xff35d515e000 — адрес освобождаемой области.

65424 — размер (≈64 KB).

15. *mprotect(0xff35d513a000, 77824, PROT\_NONE) = 0*

Запрещает доступ к определённому участку памяти. Обычно это защита между сегментами памяти, чтобы избежать ошибок.

0xff35d513a000 — адрес.

77824 — размер (≈77 KB).

PROT\_NONE — доступ запрещён.

16. *mmap(0xff35d514d000, 20480, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x19d000) = 0xff35d514d000*

Загружает сегмент данных (.data) libc.so.6.

0xff35d514d000 — адрес.

20480 — размер (20 KB).

PROT\_READ|PROT\_WRITE — доступ на чтение и запись.

3, 0x19d000 — смещение 0x19d000 (в файле libc.so.6).

17. *mmap(0xff35d5152000, 49040, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_FIXED|MAP\_ANONYMOUS, -1, 0) = 0xff35d5152000*

Выделяет память под .bss (неинициализированные переменные).

0xff35d5152000 — адрес.

49040 — размер (≈49 KB).

PROT\_READ|PROT\_WRITE — доступ на чтение и запись.

MAP\_ANONYMOUS — нет привязки к файлу.

18. *close(3) = 0*

Закрывает дескриптор 3 (libc.so.6 больше не нужен в файле).  
  
19. *set\_tid\_address(0xff35d51a7f90) = 8740*

Устанавливает адрес идентификатора потока (TLS).

0xff35d51a7f90 — адрес для хранения TID.

8740 — ID потока.

Что такое поток (Thread)? Обычно, когда ты запускаешь программу, она выполняется в одном потоке (основном). Но некоторые программы могут создавать дополнительные потоки — это позволяет делать несколько вещей одновременно.

Проблема с потоками: у всех потоков одна и та же память.

Но иногда нам нужно, чтобы у каждого потока были свои уникальные данные (например, свой номер потока или какие-то временные переменные).

TLS (Thread-Local Storage) — это механизм, который создаёт переменные, доступные только для одного потока.

set\_tid\_address(0xff35d51a7f90) = 8740 - Эта системная вызов говорит ОС: "Вот сюда записывай мой ID потока (TID)"

ОС использует это при pthread\_join и clone (создании потоков). Это важно для работы futex (системы мьютексов в Linux). При завершении потока ядро знает, куда записать статус.

20. *set\_robust\_list(0xff35d51a7fa0, 24) = 0*

Эта функция говорит ядру: "Вот список мьютексов, которые принадлежат этому потоку"

0xff35d51a7fa0 — указатель на список "робастных мьютексов" (мьютексы, которые ОС убирает при краше потока).

24 — размер структуры списка в байтах.

Мьютекс — это замок, который разрешает только одному потоку изменять данные.

Как работает?

1) Поток запрашивает мьютекс.

2) Если мьютекс свободен → поток захватывает его и работает.

3) Если мьютекс занят → поток ждёт, пока он освободится.

4) После работы поток освобождает мьютекс.

Когда мы вызываем set\_robust\_list, мы передаём список мьютексов.

Если поток умирает, не освободив мьютекс, ядро автоматически разблокирует его, чтобы не было зависания.

21. *rseq(0xff35d51a85e0, 0x20, 0, 0xd428bc00) = 0*

Эта функция используется для ускорения работы потоков, минимизируя обращения к ядру.

Разбор аргументов:

0xff35d51a85e0 — указатель на структуру rseq (register sequence).

0x20 (32 байта) — размер структуры.

0 — "флаг", не используется.

0xd428bc00 — уникальный идентификатор процессора.

22. *mprotect(0xff35d514d000, 12288, PROT\_READ) = 0*

Изменяет права доступа к памяти.

0xff35d514d000 — адрес начала памяти.

12288 (12 KB) — размер изменяемого региона.

PROT\_READ — делает память только для чтения (записывать нельзя).

23. *mprotect(0xcaa4502bf000, 4096, PROT\_READ) = 0*

Изменяет права доступа к памяти.

24. *mprotect(0xff35d51ac000, 8192, PROT\_READ) = 0*

Изменяет права доступа к памяти.

25. *prlimit64(0, RLIMIT\_STACK, NULL, {rlim\_cur=8192\*1024, rlim\_max=RLIM64\_INFINITY}) = 0*

Получает ограничения на размер стека потока. ОС следит, чтобы стек не рос бесконечно и не забил всю память.

Иногда программы меняют лимит (например, большие рекурсии).

0 — текущий процесс.

RLIMIT\_STACK — интересует именно стек.

NULL — мы не изменяем, а просто читаем лимит.

{rlim\_cur=8192\*1024, rlim\_max=RLIM64\_INFINITY} —

rlim\_cur=8MB (обычный стек — 8 мегабайт).

rlim\_max=бесконечность (максимальный размер, если нужно расширить).

26. munmap(0xff35d5196000, 66427) = 0

Освобождает память.

27. *fstat(1, {st\_mode=S\_IFCHR|0620, st\_rdev=makedev(0x88, 0), ...}) = 0*

Получает информацию о файле (в данном случае — о stdout). Перед записью программа проверяет, куда она пишет (файл? терминал?).

1 — дескриптор файла (stdout).

S\_IFCHR — это символьное устройство (экран/терминал).

0620 — права доступа (rw--w----).

28. *getrandom("\x6d\x3a\xba\x21\xb0\xc3\x96\x89", 8, GRND\_NONBLOCK) = 8*Получает случайные байты от ядра Linux.

"\x6d\x3a\xba\x21\xb0\xc3\x96\x89" – буфер, куда записаны 8 случайных байтов.

8 – количество байтов, которые нужно получить.

GRND\_NONBLOCK – флаг, который означает "не блокировать":

Если есть данные → сразу возвращает их.

Если данных нет → не ждёт, а возвращает ошибку (EAGAIN).

Результат:

Функция успешно получила 8 байтов случайных данных и записала их в буфер.

Возвращённое значение 8 говорит, что всё 8 байтов получены.

Откуда берутся случайные байты?

В Linux есть два специальных источника случайных чисел:

/dev/random – блокируется, если нет достаточной энтропии (медленный).

/dev/urandom – не блокируется, использует криптографический генератор (быстрее).

getrandom() использует тот же механизм, что /dev/urandom, но без необходимости открывать файл.

Где это используется?

Инициализация криптографии (например, TLS-соединения).

Генерация случайных идентификаторов или токенов безопасности.

29. *brk(NULL) = 0xcaa487345000*

Возвращает адрес вершины кучи.

30. *brk(0xcaa487366000) = 0xcaa487366000*

Расширяет или уменьшает размер кучи.

0xcaa487366000 – новая граница кучи (увеличена по сравнению с 0xcaa487345000).

31. *write(1, "Hello, World!\n", 14) = 14*

Записывает 14 байт в стандартный вывод (stdout).

1 — дескриптор файла (stdout = терминал).

"Hello, World!\n" — строка, которую мы выводим.

14 — длина строки в байтах.

Возвращает 14, потому что записано все 14 байт успешно.

Как это работает?

1) Программа вызывает write.

2) Ядро копирует данные в буфер вывода.

3) Драйвер терминала обрабатывает их и показывает на экране.

Почему без \n не видно?

Терминал ждёт перевод строки (\n), чтобы вывести буфер на экран.

32. *exit\_group(0) = ?*

Завершает процесс.

i. обратите внимание какие системные вызовы были вызваны в процессе исполнения программы. Чем обусловлено такое количество системных вызовов. Какой системный вызов используется для вывода “hello world”? Изучите этот вызов и разберитесь что он принимает и возвращает.

Почему так много системных вызовов?

1) Запускаем исполняемый файл → ядро должно загрузить его в память.

2) Подгружаются динамические библиотеки → libc.so.6 (так работает printf).

3) Выделяется память → стек, куча, буферы.

4) Настраиваются потоки → даже если у тебя однопоточная программа.

5) Вызывается write() → реальный системный вызов, который пишет в stdout.

6) Программа завершает работу → exit\_group().

Так что даже самая маленькая программа взаимодействует с ядром много раз.

1. Исполнение программы

*execve("./hello", ["./hello"], 0xffffe6eb5dd0 /\* 48 vars \*/) = 0*

Запускаем программу → передаём аргументы и переменные окружения.

2. Инициализация памяти

*brk(NULL) = 0xcaa487345000*

*mmap(NULL, 8192, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0xff35d51a7000*

brk(NULL) узнаёт границу кучи.

mmap() создаёт память для структуры управления потоками (pthread).

3. Подгрузка динамических библиотек

*faccessat(AT\_FDCWD, "/etc/ld.so.preload", R\_OK) = -1 ENOENT (No such file or directory)*

*openat(AT\_FDCWD, "/etc/ld.so.cache", O\_RDONLY|O\_CLOEXEC) = 3*

*fstat(3, {st\_mode=S\_IFREG|0644, st\_size=66427, ...}) = 0*

*mmap(NULL, 66427, PROT\_READ, MAP\_PRIVATE, 3, 0) = 0xff35d5196000*

*close(3) = 0*

Ищем /etc/ld.so.preload (нет такого файла).

Открываем /etc/ld.so.cache (кешированный список библиотек).

Загружаем libc.so.6 (именно здесь printf()!).

openat(AT\_FDCWD, "/lib/aarch64-linux-gnu/libc.so.6", O\_RDONLY|O\_CLOEXEC) = 3

Без libc.so.6 программа не сможет вызвать printf()!

4. Размещение libc.so.6 в памяти

*mmap(NULL, 1892240, PROT\_NONE, MAP\_PRIVATE|MAP\_ANONYMOUS|MAP\_DENYWRITE, -1, 0) = 0xff35d4fa0000*

*mmap(0xff35d4fa0000, 1826704, PROT\_READ|PROT\_EXEC, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0) = 0xff35d4fa0000*

Выделяем память.

Загружаем libc.so.6 в процесс.

5. Настройка потоков

*set\_tid\_address(0xff35d51a7f90) = 8740*

*set\_robust\_list(0xff35d51a7fa0, 24) = 0*

Даже если у нас нет pthread, glibc всегда включает поддержку потоков (на случай fork(), exec(), malloc() и других вещей).

**6. Вывод "Hello, World!"**

Это самый главный вызов. Именно он реально пишет в терминал.

1 → файловый дескриптор (1 = stdout).

"Hello, World!\n" → что выводим.

14 → длина строки (включая \n).

14 в результате → значит, все 14 байтов записаны успешно.

Как работает printf()?

printf() сам не вызывает write() напрямую. Он сначала пишет в буфер stdio. Когда буфер заполняется или вызывается \n, glibc делает write().

Синтаксис write():

*ssize\_t write(int fd, const void \*buf, size\_t count);*

Аргументы:

fd → файловый дескриптор (куда писать).

1 — стандартный вывод (stdout).

2 — стандартный поток ошибок (stderr).

Можно писать в файл, сокет и т. д.

buf → указатель на буфер с данными.

У нас "Hello, World!\n".

count → количество байтов для записи.

"Hello, World!\n" состоит из 14 байтов.

Bозвращает:

Число фактически записанных байтов.

Если меньше count → была ошибка или буфер заполнился.

-1 если ошибка (например, fd недоступен).

ii. используйте этот сискол в программе hello world вместо printf(). Убедитесь что этот вызов присутствует в выводе strace.

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <errno.h>

#include <unistd.h>

#define ERROR (-1)

int main(void) {

char buffer[] = "Hello, World!\n";

ssize\_t result;

ssize\_t written\_bytes = 0;

ssize\_t length = sizeof(buffer);

while (written\_bytes < length) {

result = write(STDOUT\_FILENO, buffer + written\_bytes, length - written\_bytes);

if (result == ERROR) {

fprintf(stderr, "Error: %s\n", strerror(errno));

return EXIT\_FAILURE;

}

written\_bytes += result;

}

return EXIT\_SUCCESS;

}

*strace ./hello\_write*

*execve("./hello\_write", ["./hello\_write"], 0xffffe39b3a50 /\* 48 vars \*/) = 0*

*brk(NULL) = 0xb609f832b000*

*mmap(NULL, 8192, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0xecb9f308e000*

*faccessat(AT\_FDCWD, "/etc/ld.so.preload", R\_OK) = -1 ENOENT (No such file or directory)*

*openat(AT\_FDCWD, "/etc/ld.so.cache", O\_RDONLY|O\_CLOEXEC) = 3*

*fstat(3, {st\_mode=S\_IFREG|0644, st\_size=66427, ...}) = 0*

*mmap(NULL, 66427, PROT\_READ, MAP\_PRIVATE, 3, 0) = 0xecb9f307d000*

*close(3) = 0*

*openat(AT\_FDCWD, "/lib/aarch64-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\267\0\1\0\0\0\360\206\2\0\0\0\0\0"..., 832) = 832*

*fstat(3, {st\_mode=S\_IFREG|0755, st\_size=1722920, ...}) = 0*

*mmap(NULL, 1892240, PROT\_NONE, MAP\_PRIVATE|MAP\_ANONYMOUS|MAP\_DENYWRITE, -1, 0) = 0xecb9f2e87000*

*mmap(0xecb9f2e90000, 1826704, PROT\_READ|PROT\_EXEC, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0) = 0xecb9f2e90000*

*munmap(0xecb9f2e87000, 36864) = 0*

*munmap(0xecb9f304e000, 28560) = 0*

*mprotect(0xecb9f302a000, 77824, PROT\_NONE) = 0*

*mmap(0xecb9f303d000, 20480, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x19d000) = 0xecb9f303d000*

*mmap(0xecb9f3042000, 49040, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_FIXED|MAP\_ANONYMOUS, -1, 0) = 0xecb9f3042000*

*close(3) = 0*

*set\_tid\_address(0xecb9f308efb0) = 88400*

*set\_robust\_list(0xecb9f308efc0, 24) = 0*

*rseq(0xecb9f308f600, 0x20, 0, 0xd428bc00) = 0*

*mprotect(0xecb9f303d000, 12288, PROT\_READ) = 0*

*mprotect(0xb609c7c3f000, 4096, PROT\_READ) = 0*

*mprotect(0xecb9f3093000, 8192, PROT\_READ) = 0*

*prlimit64(0, RLIMIT\_STACK, NULL, {rlim\_cur=8192\*1024, rlim\_max=RLIM64\_INFINITY}) = 0*

*munmap(0xecb9f307d000, 66427) = 0*

*write(1, "Hello, World!\n\0", 15Hello, World!*

*) = 15*

*exit\_group(0) = ?*

*+++ exited with 0 +++*

Программа использует системный вызов write() вместо printf() для вывода "Hello, World!\n". Мы подтвердили, что write() вызывается в strace:

*write(1, "Hello, World!\n\0", 15) = 15*

iii. напишите свою обертку над этим сисколом. Для этого используйте функцию syscall() из libc. Также проверьте вывод strace.

#include <errno.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/syscall.h>

#include <unistd.h>

#define ERROR (-1)

ssize\_t my\_write(unsigned int file\_descriptor, const char \*buffer, size\_t length) {

return syscall(SYS\_write, file\_descriptor, buffer, length);

}

int main(void) {

char buffer[] = "Hello, World!\n";

ssize\_t result;

ssize\_t written\_bytes = 0;

ssize\_t length = sizeof(buffer);

while (written\_bytes < length) {

result = my\_write(STDOUT\_FILENO, buffer + written\_bytes, length - written\_bytes);

if (result == ERROR) {

fprintf(stderr, "Error: %s\n", strerror(errno));

return EXIT\_FAILURE;

}

written\_bytes += result;

}

return EXIT\_SUCCESS;

}

b. Запустите под strace команду ‘wget kernel.org’ (если нет wget, используйте curl). Получите статистику использования системных вызовов порожденным процессом.

*strace -c wget kernel.org*

*--2025-03-06 15:42:13-- http://kernel.org/*

*Resolving kernel.org (kernel.org)... 139.178.84.217*

*Connecting to kernel.org (kernel.org)|139.178.84.217|:80... connected.*

*HTTP request sent, awaiting response... 301 Moved Permanently*

*Location: https://kernel.org/ [following]*

*--2025-03-06 15:42:13-- https://kernel.org/*

*Connecting to kernel.org (kernel.org)|139.178.84.217|:443... connected.*

*HTTP request sent, awaiting response... 200 OK*

*Length: 16834 (16K) [text/html]*

*Saving to: ‘index.html’*

*index.html 100%[===================>] 16.44K --.-KB/s in 0.001s*

*2025-03-06 15:42:15 (12.1 MB/s) - ‘index.html’ saved [16834/16834]*

*% time seconds usecs/call calls errors syscall*

*------ ----------- ----------- --------- --------- ----------------*

*12.93 0.001203 34 35 write*

*11.42 0.001062 8 119 7 read*

*8.59 0.000799 20 39 mmap*

*7.34 0.000683 62 11 pselect6*

*6.84 0.000636 28 22 mprotect*

*6.75 0.000628 12 51 25 openat*

*6.71 0.000624 124 5 2 connect*

*5.45 0.000507 25 20 12 newfstatat*

*4.56 0.000424 12 33 ioctl*

*3.82 0.000355 11 30 close*

*3.66 0.000340 170 2 getdents64*

*3.58 0.000333 15 21 munmap*

*3.27 0.000304 11 27 getpgid*

*2.68 0.000249 8 30 fstat*

*2.29 0.000213 10 21 brk*

*2.04 0.000190 38 5 socket*

*1.44 0.000134 8 15 getpid*

*1.40 0.000130 14 9 fcntl*

*0.74 0.000069 2 25 futex*

*0.72 0.000067 22 3 recvfrom*

*0.67 0.000062 20 3 ppoll*

*0.62 0.000058 8 7 rt\_sigaction*

*0.57 0.000053 53 1 sendmmsg*

*0.31 0.000029 29 1 utimensat*

*0.27 0.000025 12 2 getrandom*

*0.26 0.000024 12 2 rt\_sigprocmask*

*0.16 0.000015 15 1 set\_tid\_address*

*0.16 0.000015 15 1 set\_robust\_list*

*0.16 0.000015 15 1 rseq*

*0.15 0.000014 14 1 ftruncate*

*0.14 0.000013 13 1 prlimit64*

*0.11 0.000010 10 1 getgroups*

*0.08 0.000007 3 2 statfs*

*0.06 0.000006 2 3 lseek*

*0.03 0.000003 3 1 flock*

*0.03 0.000003 3 1 getuid*

*0.00 0.000000 0 1 1 faccessat*

*0.00 0.000000 0 2 setsockopt*

*0.00 0.000000 0 1 execve*

*------ ----------- ----------- --------- --------- ----------------*

*100.00 0.009302 16 556 47 total*

Команда strace -c wget kernel.org используется для отслеживания системных вызовов, выполняемых программой wget при скачивании файла с сайта kernel.org.

**Чтение данных**: Когда программа, такая как wget, скачивает файл из интернета, она сначала выполняет сетевой запрос (например, HTTP-запрос) к серверу, чтобы получить данные. На этом этапе данные не "считываются" из файла, а загружаются через сетевое соединение. Но именно системный вызов read используется для считывания данных, полученных от сервера, из сетевого сокета.

read: В этом контексте используется для получения данных из сетевого соединения (сокета). Программа вызывает read для того, чтобы читать данные, полученные по сети. Эти данные могут быть буферизованы в памяти, и read считывает их из этого буфера в локальную память программы (например, в буфер в оперативной памяти).

**Запись данных**: После того как данные были считаны из сетевого сокета, программа должна их куда-то записать. В случае с wget, данные записываются в файл на диск (или в стандартный вывод, если файл не указан). Для этого используется системный вызов write.

write: Этот системный вызов записывает данные в файл, который может быть как обычным файлом на диске, так и стандартным выводом (например, если вы указали в командной строке wget выводить данные на экран). В случае скачивания файла это обычно запись в файл, куда сохраняются загруженные данные.

Процесс скачивания файла:

1) wget устанавливает сетевое соединение с сервером, чтобы запросить файл.

2) Программа использует системный вызов read для получения данных из сокета (сервера).

3) Полученные данные затем записываются в локальный файл с помощью write.

Основные системные вызовы:

1. Системный вызов write:

12.93% time — 12.93% общего времени процесса ушло на выполнение системного вызова write.

0.001203 seconds — общее время, потраченное на write.

34 usecs/call — среднее время одного вызова write составляет 34 микросекунды.

35 calls — вызов write был сделан 35 раз.

write в данном случае используется для записи данных (например, заголовков HTTP, информации о прогрессе или полученных данных) в стандартный вывод или файл.

2. Системный вызов read:

11.42% time — 11.42% времени потрачено на выполнение системного вызова read, который используется для чтения данных из сети.

0.001062 seconds — общее время для всех операций чтения.

119 calls — системный вызов read был вызван 119 раз.

7 errors — при выполнении чтений возникло 7 ошибок.

3. Системный вызов mmap:

8.59% time — на выполнение вызова mmap ушло 8.59% времени.

0.000799 seconds — общее время на маппинг памяти.

39 calls — системный вызов mmap был вызван 39 раз.

4. Системный вызов openat:

6.75% time — 6.75% времени затрачено на открытие файлов с использованием системного вызова openat.

0.000628 seconds — время, потраченное на открытие файлов.

51 calls — вызвано 51 обращение к openat.

5. Системный вызов connect:

6.71% time — 6.71% времени ушло на подключение к удаленному серверу с использованием connect.

124 usecs/call — среднее время одного вызова соединения составляет 124 микросекунды.

5 calls — вызвано 5 соединений.  
write (12.93%): Запись данных в файл или в устройство (например, в терминал или в сокет).

read (11.42%): Чтение данных из файла или устройства (например, из сокета, файла или терминала).

mmap (8.59%): Отображение файла в память или выделение памяти. Используется для работы с файлами или для выделения области памяти.

pselect6 (7.34%): Ожидание события на одном или нескольких файловых дескрипторах (например, для работы с сокетами или файлами).

mprotect (6.84%): Изменение прав доступа к памяти, например, для установки прав чтения/записи/исполнения.

openat (6.75%): Открытие файла, с возможностью указания директории относительно которой файл будет открыт.

connect (6.71%): Установка соединения с удалённым сервером через сокет (например, для соединения с сервером по протоколу TCP).

newfstatat (5.45%): Получение информации о файле (например, размер файла, дата изменения и другие атрибуты).

ioctl (4.56%): Управление параметрами устройств (например, изменение характеристик сокетов, терминалов или других устройств).

close (3.82%): Закрытие файлового дескриптора.

getdents64 (3.66%): Чтение содержимого директории (возвращает список файлов в директории).

munmap (3.58%): Освобождение ранее отображенной в память области, например, после работы с файлом через mmap.

getpgid (3.27%): Получение идентификатора группы процессов для данного процесса.

fstat (2.68%): Получение информации о файле, как и newfstatat, но без указания директории.

brk (2.29%): Управление выделением памяти для процесса (работа с кучей).

socket (2.04%): Создание нового сокета для сетевых операций.

getpid (1.44%): Получение идентификатора текущего процесса.

fcntl (1.40%): Операции с файловыми дескрипторами, например, изменение флагов или блокировка.

futex (0.74%): Работа с блокировками и синхронизацией потоков.

recvfrom (0.72%): Чтение данных из сокета.

ppoll (0.67%): Ожидание событий на файлах или сокетах с использованием сигналов.

rt\_sigaction (0.62%): Управление обработчиками сигналов (например, установление функции-обработчика для определённого сигнала).

sendmmsg (0.57%): Отправка нескольких сообщений через сокет.

utimensat (0.31%): Изменение временных меток файлов.

getrandom (0.27%): Получение случайных чисел (обычно для криптографических операций).

rt\_sigprocmask (0.26%): Изменение маски сигналов (например, блокировка сигналов).

set\_tid\_address (0.16%): Установка адреса идентификатора потока (для многозадачности).

set\_robust\_list (0.16%): Установка списка защищённых операций, которые должны быть завершены в случае аварийного завершения

процесса.

rseq (0.16%): Программный интерфейс для поддержки последовательной обработки данных (особенно в многозадачности).

ftruncate (0.15%): Обрезание файла до указанной длины.

prlimit64 (0.14%): Управление ресурсами процесса, например, лимитами памяти.

getgroups (0.11%): Получение групповых идентификаторов текущего пользователя.

statfs (0.08%): Получение информации о файловой системе.

lseek (0.06%): Перемещение указателя текущей позиции в файле.

flock (0.03%): Блокировка файлов для синхронизации доступа.

getuid (0.03%): Получение идентификатора текущего пользователя.

faccessat (0.00%): Проверка доступа к файлу (можно использовать для проверки прав на чтение/запись).

setsockopt (0.00%): Установка опций для сокетов (например, для настройки таймаутов или флагов).

execve (0.00%): Запуск нового процесса.

**ДОП. ВОПРОСЫ**

1. Зачем нужен buffer в printf?

Буферизация – это промежуточное хранение данных перед их обработкой или передачей. Хотя без буфера можно обойтись, это приведёт к значительному снижению производительности и ряду проблем.

1)Системные вызовы – дорогая операция

Каждый вызов write() или read() – это системный вызов (syscall), который требует переключения из пользовательского режима (user space) в режим ядра (kernel space). Это занимает время.

Без буфера: Каждый символ требует отдельного вызова write(), что сильно замедляет программу.

С буфером: Мы уменьшаем количество write() до одного вызова.

2) Оптимизация работы с диском

Запись на диск происходит блоками (например, 4 КБ), а не байт за байтом. Если писать по 1 байту без буфера, диск будет работать неэффективно.

Без буфера: Диск будет постоянно обращаться к физическим секторам.

С буфером: Данные сначала копятся в оперативной памяти, а затем одной операцией записываются на диск, что намного быстрее.

Функция printf() использует буфер внутри стандартной библиотеки (stdio). Поэтому вызовы:

printf("Hello");

printf(" World!");

не обязательно приведут к немедленному вызову write(). printf() копит данные в буфере и вызывает write() только:

- когда буфер заполнен,

- при вызове fflush(stdout),

- при exit().

2. Чем отличается printf от write?

1) printf

Относится к библиотечным функциям C (стандартная библиотека stdio.h).

Предоставляет удобный форматированный вывод (например, можно выводить числа в разных системах счисления, строки, символы и т. д.).

Внутри использует буферизацию, то есть данные могут накапливаться перед фактическим выводом.

Вызывает системный вызов write, но перед этим выполняет обработку данных (форматирование, буферизацию).

2) write

Это системный вызов (syscall), который обращается напрямую к ядру ОС.

Определён в unistd.h и работает с файловыми дескрипторами (fd).

Не выполняет форматирование, просто записывает сырые байты.

Не использует стандартные буферы ввода-вывода C, но может использовать буферизацию ядра.

3. Когда мы можем быть уверены, что данные точно попали в файл?

Данные считаются записанными в файл, когда они гарантированно хранятся на диске, а не просто находятся в буферах. Однако, из-за кэширования операционной системы, даже после write() данные могут быть только в памяти (кэше) и ещё не записаны на диск.

1) Использование fsync(fd)

После write(), вызов fsync(fd) принудительно записывает данные из кэша ОС на диск.

2) Использование O\_SYNC при открытии файла

Флаг O\_SYNC заставляет write() сразу записывать данные на диск, минуя кэш ОС.

3) Использование fflush(FILE \*) для stdio

Если записываем через fwrite() или printf(), данные сначала попадают в буфер stdio. Чтобы их отправить в ОС (но не обязательно сразу на диск), нужно вызвать fflush(). Но fflush() не гарантирует запись на диск, он только передаёт данные ядру.

4) Использование fsync() после fflush()

Чтобы данные точно попали на диск, нужно сначала сбросить буфер stdio, а затем вызвать fsync()