**ОБЩАЯ ТЕОРИЯ**

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

**С помощью ptrace МОЖНО:**

1. Читать и изменять память и регистры трассируемого процесса.

2. Перехватывать системные вызовы и сигналы.

3. Останавливать и возобновлять выполнение трассируемого процесса.

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

**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 \_РОКЕТЕХТ : 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.

**ЗАДАНИЕ**

Разберитесь как работает системный вызов ptrace(2) и напишите программу, которая породит процесс и выведет все системные вызовы дочернего процесса.

#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) {

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(void) {

pid\_t child\_pid;

child\_pid = fork(); // родительсий процесс получает pid дочернего процесса, дочерний процесс получает 0

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. 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.

Вывод программы:

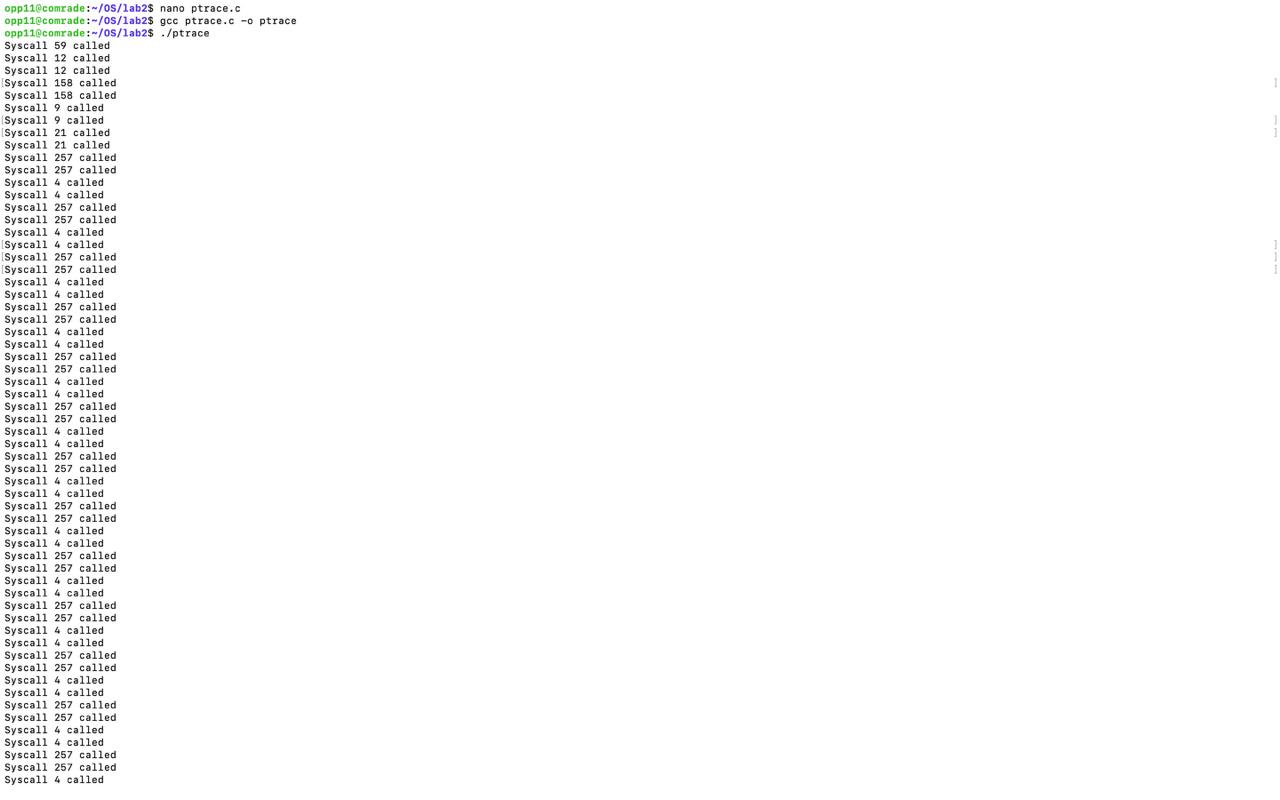


Таблица системных вызовов:

<https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md#x86_64-64_bit>

Вызовы которые были выведены:

Syscall 59 called - **execve**, для загрузки и запуска программы в процессе. Был вызван для запуска /bin/ls

Syscall 12 called - **brk**, для изменения размеры кучи процесса. Обычно вызывается при выделении памяти

Syscall 158 called - **arch\_prctl**, настройка архитектурных специфичных параметров процесса, таких как установка архитектурных особенностей. Он может задавать обработчик ошибок для выполнения некоторых инструкций, которые зависят от архитектуры процессора.

Syscall 9 called - **mmap**, отображение файла в память, выделение памяти.

Syscall 21 called - **access**, проверка прав доступа к файлу или директории.

Syscall 257 called - **openat**, открывает файл или директорию

Syscall 4 called - **stat**, получение информации о файле или директории (размер файла, права доступа и т.д.)

Syscall 5 called - **fstat**, получает информацию о файле (размер, права доступа и т.д.). работает аналогично stat, но вместо пути к файлу, он использует файловый дескриптор. Это полезно, когда файл уже открыт и требуется получить его метаданные без необходимости снова искать его по пути.

Syscall 11 called - **munmap**, освобождение области памяти, которая была выделена ранее с помощью mmap

Syscall 218 called - **set\_tid\_address**, установка адреса, по которому будет сохранен идентификатор потока (TID). Это нужно для многозадачности и многопоточности, когда необходимо отслеживать идентификаторы потоков в группе.

Syscall 273 called - **set\_robust\_list**, настройка списка "робустных" блокировок в многозадачных и многопоточных программах. Эти блокировки гарантируют корректное поведение в случае, если поток, владеющий блокировкой, завершится неожиданно.

Syscall 13 called - **rt\_sigaction**, установка обработчиков сигналов в реальном времени. Это позволяет программе перехватывать и обрабатывать асинхронные события, такие как сигналы от операционной системы или других процессов.

Syscall 14 called - **rt\_sigprocmask**, блокировка или разблокировка сигналов. Это важно для контроля над асинхронными событиями, чтобы избежать их обработки в неподобающий момент.

Syscall 302 called - **prlimit64**, получение или установка ограничений ресурсов процесса, таких как максимальный размер стека или количество открытых файловых дескрипторов.

Syscall 137 called - **statfs**, получение информации о файловой системе, такой как доступное пространство на диске, максимальный размер файлов и другие параметры файловой системы.

Syscall 3 called - **close**, закрытие открытого файлового дескриптора

Syscall 202 called - **futex**, реализация блокировок и синхронизация потоков в многозадачных системах. Это низкоуровневый механизм синхронизации, который позволяет эффективно управлять потоками.

Syscall 16 called - **ioctl**, выполнение различных управляющих операций с файловыми дескрипторами, таких как настройка параметров устройства или получение статуса устройства.

Syscall 0 called - **read**, чтение данных из файла или устройства

Syscall 10 called - **mprotect**, изменение прав доступа к области памяти

Syscall 17 called - **pread64**, чтение данных из файла на заданной позиции, не изменяя текущую позицию в файле

Syscall 217 called - **getdents64**, чтение записей из директории в 64-битном формате

Syscall 1 called - **write**, записывает данные в файловый дескриптор. Был вызван для вывода содержимого директории на экран

Syscall 231 called - **exit\_group**, завершает выполнение всей группы процессов (в отличие от exit, который завершает только текущий поток)

1. wait()

Функция **wait(&status)** в родительском процессе *приостанавливает его выполнение*, пока любой из *дочерних процессов не изменит своё состояние* (например, завершится или будет остановлен каким-либо сигналом, в том числе от ptrace).

Возвращает PID дочернего процесса, который изменил своё состояние.

В status (передаётся по указателю) кодируется причина остановки дочернего процесса (завершение, сигнал, остановка).

status — это битовая маска (число, в котором закодирована информация). Она может содержать:

• Код завершения (если процесс завершился сам).

• Номер сигнала (если процесс был убит сигналом).

• Флаги остановки (если процесс был остановлен, например, SIGSTOP).

Чтобы извлечь эту информацию, используют макросы:

WIFEXITED(status)

Возвращает true, если процесс завершился нормально (через exit() или return).

WEXITSTATUS(status)

Если WIFEXITED вернул true, то этот макрос даёт код завершения (то, что передали в exit()).

WIFSIGNALED(status)

Возвращает true, если процесс был убит сигналом (например, SIGKILL).

WTERMSIG(status)

Если WIFSIGNALED вернул true, то этот макрос даёт номер сигнала, который убил процесс.

WIFSTOPPED(status)

Возвращает true, если процесс был остановлен (например, SIGSTOP или ptrace).

WSTOPSIG(status)

Если WIFSTOPPED вернул true, то этот макрос даёт номер сигнала, который остановил процесс.

1 Дочерний процесс (ls):

◦ Делает первый syscall → останавливается.

◦ Родитель получает управление через wait(&status).

◦ WIFEXITED(status) = false (просто остановка).

◦ Родитель смотрит регистры (PTRACE\_GETREGS), видит номер syscall.

◦ Родитель говорит: "Продолжай" (PTRACE\_SYSCALL).

2 Дочерний процесс продолжает выполнение:

◦ Syscall выполняется в ядре.

◦ После выполнения снова останавливается.

◦ Родитель опять получает управление.

◦ WIFEXITED(status) всё ещё false.

◦ Родитель может проверить результат syscall (например, rax).

◦ Снова говорит "Продолжай" (PTRACE\_SYSCALL).

3 Так продолжается, пока ls не завершится:

◦ Когда ls делает exit(), родитель получает статус WIFEXITED(status) = true.

◦ Срабатывает break, и цикл трассировки завершается.

Как связаны SIGKILL, SIGSTOP и status?

• Если процесс убит сигналом SIGKILL (например, kill -9 PID), то:

◦ WIFSIGNALED(status) → true

◦ WTERMSIG(status) → 9 (потому что SIGKILL = 9)

• Если процесс остановлен сигналом SIGSTOP (например, kill -19 PID), то:

◦ WIFSTOPPED(status) → true

◦ WSTOPSIG(status) → 19 (потому что SIGSTOP = 19)

Когда процесс трассируется через ptrace, ядро Linux приостанавливает его выполнение на каждом системном вызове и передаёт управление отладчику (родительскому процессу).

1. Как ptrace ловит syscall'ы?

Механизм №1: SIGTRAP (сигнал трассировки)

Когда трассируемый процесс (child\_pid) выполняет системный вызов, ядро:

1 Приостанавливает процесс перед входом в syscall (до выполнения кода ядра).

2 Отправляет сигнал SIGTRAP (код 5), но не как обычный сигнал, а как специальное событие ptrace.

3 Родительский процесс получает уведомление через wait(&status).

4 Проверив status, родитель видит, что процесс остановлен (WIFSTOPPED) из-за SIGTRAP.

Механизм №2: PTRACE\_SYSCALL (явный запрос на остановку)

Когда родитель делает:

ptrace(PTRACE\_SYSCALL, child\_pid, NULL, NULL);

Он говорит ядру:

«Запусти дочерний процесс, но останови его при входе в syscall и при выходе из него».

Это приводит к двум остановкам на каждый syscall:

1 Перед выполнением (можно прочитать аргументы syscall из регистров).

2 После выполнения (можно прочитать возвращаемое значение в rax).

2. Какой сигнал приходит в status?

Когда процесс останавливается на syscall, wait(&status) получает не обычный SIGTRAP, а специальный статус остановки ptrace.

Как проверить, что остановка из-за syscall?

1 WIFSTOPPED(status) → true (процесс остановлен).

2 WSTOPSIG(status) → SIGTRAP (но это не обычный SIGTRAP, а сигнал от ptrace).  
  
Полный цикл обработки syscall'ов в твоём коде

1 Родитель запускает дочерний процесс с PTRACE\_TRACEME.

2 Дочерний процесс (ls) делает первый syscall → ядро останавливает его и отправляет SIGTRAP.

3 Родитель ловит это через wait(&status):

◦ WIFEXITED(status) → false (процесс не завершился),

◦ WIFSTOPPED(status) → true (остановлен),

◦ WSTOPSIG(status) → 5 (SIGTRAP).

4 Родитель смотрит регистры (PTRACE\_GETREGS), видит номер syscall (orig\_rax).

5 Родитель говорит: ptrace(PTRACE\_SYSCALL, child\_pid, NULL, NULL); // "Продолжай, но остановись на следующем syscall"

6 Дочерний процесс выполняет syscall, затем снова останавливается (на выходе).

7 Родитель снова получает управление, может прочитать результат syscall (в rax).

8 Цикл повторяется, пока ls не завершится (WIFEXITED(status) = true).  
  
**SIGTRAP** — это один из стандартных сигналов в Unix/Linux, который имеет номер 5.

Он расшифровывается как "Trap", что дословно означает "ловушка" или "захват".

Этот сигнал используется, чтобы остановить выполнение процесса и уведомить отладчик (например, gdb, strace или ptrace), что произошло что-то интересное — например:

- Процесс дошёл до точки останова (breakpoint).

- Произошёл системный вызов (если мы явно хотим отслеживать это через ptrace).

- Выполнена команда int 3 в ассемблере (это команда, вставляемая отладчиком).

- Выполнена трассировка через ptrace(PTRACE\_SYSCALL).

**Что делает SIGTRAP?**

Когда процесс получает SIGTRAP, он останавливается и сообщает родительскому процессу (например, отладчику), что "что-то произошло".

Отладчик (или твоя программа) может затем:

- Посмотреть регистры, память, стек и т. д.

- Принять решение: продолжать выполнение, изменить что-то, остановиться и т.д.

**Почему SIGTRAP, но "необычный"?**

Когда ты используешь ptrace с PTRACE\_SYSCALL, ты говоришь ядру:

«Останови ребёнка каждый раз перед входом и после выхода из системного вызова».

Каждый раз, когда процесс достигает такого "остановочного" места (например, перед write() или после read()), ядро:

- Останавливает процесс

- Отсылает родителю сигнал SIGTRAP

- Родитель (твоя программа) через wait() видит, что процесс остановлен (WIFSTOPPED)

- И может, например, получить регистры, прочитать номер системного вызова и т. д.

⚠️ Но это не "обычный" SIGTRAP.

Это внутренний механизм ptrace — это не отладочная ловушка, установленная самим процессом.

То есть: процесс не сам себе послал SIGTRAP, это ядро сообщает родителю, что ptrace-действие произошло (вход/выход из syscall).  
  
**В классическом случае**:

- Программа сама вызывает ловушку — например, через ассемблерную инструкцию int 3, которую, например, вставляет gdb при установке точки останова (breakpoint).

- Это вызывает исключение (trap) в процессоре → процесс останавливается, ядро шлёт ему SIGTRAP.

Пример: ты пишешь программу и вручную вставляешь asm("int $3");, чтобы остановить её при отладке. Это обычный SIGTRAP, его можно поймать, изменить, игнорировать и т.п.

**А что происходит при PTRACE\_SYSCALL?**

Когда ты делаешь:

ptrace(PTRACE\_SYSCALL, child\_pid, NULL, NULL);Ты говоришь ядру: «Я хочу, чтобы ты приостановил исполняемый процесс каждый раз:

- перед тем как он входит в системный вызов

- и сразу после выхода из системного вызова»

И что делает ядро?

- Дочерний процесс продолжает работать.

- Он доходит до системного вызова (например, write()).

- Прямо перед выполнением системного вызова ядро останавливает процесс.

- Ядро посылает родительскому процессу (ptrace-отладчику) сигнал SIGTRAP.

- Родитель получает SIGTRAP, вызывает wait(), видит, что ребёнок остановился → может прочитать регистры, номер системного вызова (orig\_rax), и т.д.

- Потом родитель снова вызывает PTRACE\_SYSCALL — и история повторяется, но теперь уже после выхода из syscall.

**А что происходит при PTRACE\_SYSCALL?**

Когда ты делаешь:

ptrace(PTRACE\_SYSCALL, child\_pid, NULL, NULL);Ты говоришь ядру: «Я хочу, чтобы ты приостановил исполняемый процесс каждый раз:

- перед тем как он входит в системный вызов

- и сразу после выхода из системного вызова»

И что делает ядро?

- Дочерний процесс продолжает работать.

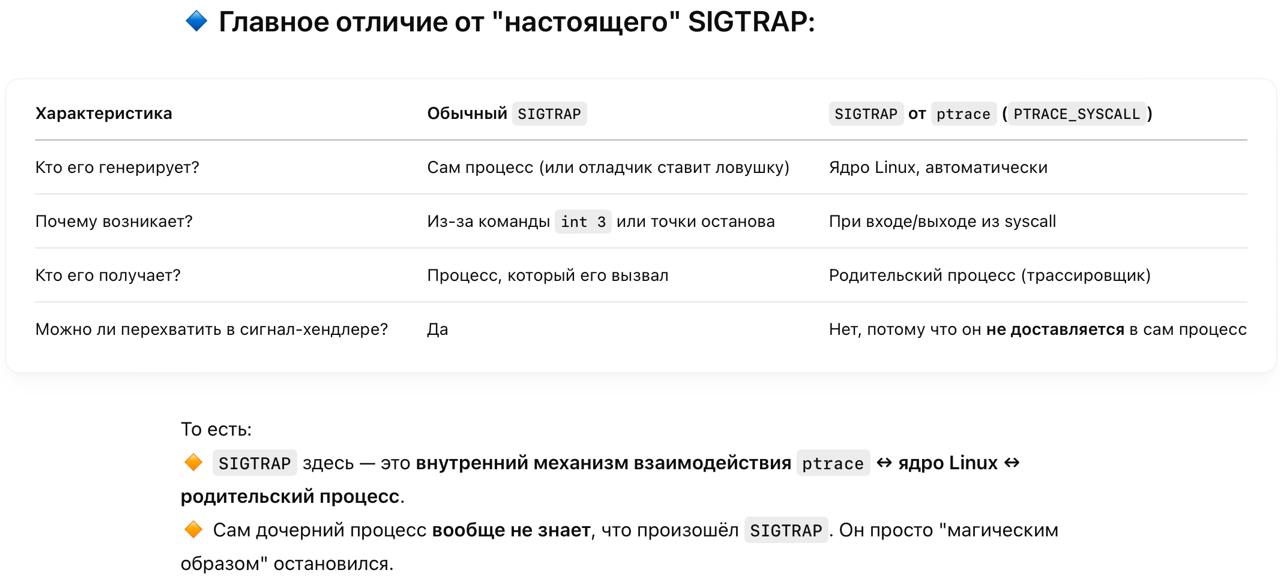
- Он доходит до системного вызова (например, write()).

- Прямо перед выполнением системного вызова ядро останавливает процесс.

- Ядро посылает родительскому процессу (ptrace-отладчику) сигнал SIGTRAP.

- Родитель получает SIGTRAP, вызывает wait(), видит, что ребёнок остановился → может прочитать регистры, номер системного вызова (orig\_rax), и т.д.

- Потом родитель снова вызывает PTRACE\_SYSCALL — и история повторяется, но теперь уже после выхода из syscall.



**Аналоги ptrace**

1. strace (System Call Tracer)

Тип: Утилита командной строки

Назначение: Трассировка системных вызовов и сигналов.

Как работает: Использует ptrace внутри, но предоставляет удобный интерфейс.

2. ltrace

Показывает, какие функции из библиотек (например, libc) вызывает программа — выделение памяти, работа с файлами и т.д.

3. eBPF (bpftrace/bcc) — «Супер-микроскоп для Linux»

Позволяет следить за системой в реальном времени без тормозов. Можно писать свои мини-программы для анализа.

4. perf trace — «Профилировщик»

Показывает, на что система тратит время (системные вызовы, работа процессора и т.д.).

5. gdb — «Отладчик для хакеров»

Позволяет «заморозить» программу в любой момент и посмотреть, что внутри:

Какие переменные сейчас в памяти.

На какой строке кода программа остановилась.

1 «Что-то не работает» → strace или ltrace (просто запустить и посмотреть логи).

2 «Программа падает» → gdb (посмотреть стек вызовов).

3 «Сервер тормозит» → perf или eBPF (найти узкое место).

Немного усовершенствованный код, где выводится не только номер системного вызова, но и аргументы и возвращаемое значение  
#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>

#include <errno.h>

**void** trace\_syscalls(pid\_t child\_pid) {

**int** status;

**int** in\_syscall = 0;

**struct** user\_regs\_struct regs;

waitpid(child\_pid, &status, 0);

ptrace(PTRACE\_SETOPTIONS, child\_pid, 0, PTRACE\_O\_TRACESYSGOOD);

**while** (1) {

ptrace(PTRACE\_SYSCALL, child\_pid, 0, 0);

waitpid(child\_pid, &status, 0);

**if** (WIFEXITED(status)) {

**break**;

}

**if** (WIFSTOPPED(status) && (WSTOPSIG(status) & 0x80)) {

ptrace(PTRACE\_GETREGS, child\_pid, **NULL**, &regs);

**if** (in\_syscall == 0) {

printf("\n[Syscall Enter] %lld(", regs.orig\_rax);

printf("rdi: 0x%llx, rsi: 0x%llx, rdx: 0x%llx, r10: 0x%llx, r8: 0x%llx, r9: 0x%llx)\n",

regs.rdi, regs.rsi, regs.rdx, regs.r10, regs.r8, regs.r9);

in\_syscall = 1;

} **else** {

printf("[Syscall Exit] return: 0x%llx (%lld)\n", regs.rax, regs.rax);

in\_syscall = 0;

}

}

}

}

**int** main(**void**) {

pid\_t child\_pid = fork();

**if** (child\_pid == 0) {

ptrace(PTRACE\_TRACEME, 0, **NULL**, **NULL**);

raise(SIGSTOP);

execl("/bin/ls", "ls", **NULL**);

perror("execl");

exit(1);

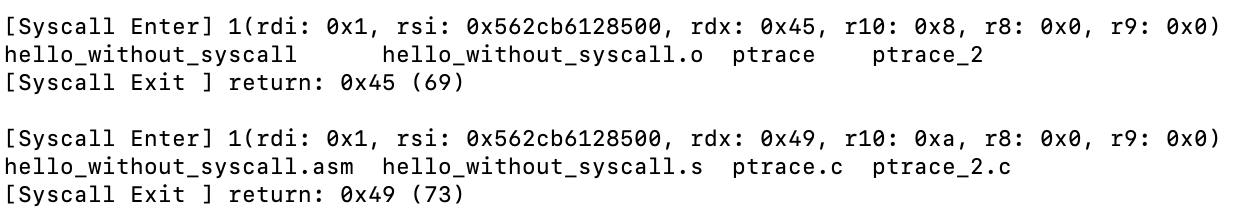
} **else** {

trace\_syscalls(child\_pid);

}

**return** 0;

}



**regs.orig\_rax**: Это регистр, в который помещается номер системного вызова. Например, если это write, то значение будет равно 1 (для 64-битных Linux-систем).

regs.rdi, regs.rsi, regs.rdx, regs.r10, regs.r8, regs.r9: Эти регистры содержат аргументы системных вызовов. В 64-битных системах, системные вызовы передают свои аргументы через регистры в следующем порядке:

rdi: 1-й аргумент

rsi: 2-й аргумент

rdx: 3-й аргумент

r10: 4-й аргумент

r8: 5-й аргумент

r9: 6-й аргумент

**regs.rax**: Этот регистр содержит результат системного вызова. Если это системный вызов типа write, то он будет содержать количество записанных байтов. Если системный вызов завершился с ошибкой, то в нем может быть код ошибки.