Лабораторная работа №1. Системные вызовы. Работа с процессами.

Цель работы: освоить механизм работы системных вызовов, научиться создавать дочерние процессы и изучить способы взаимодействия родительского и дочернего процесса.

Теоретические сведения

Механизм системных вызовов

Пользовательские приложения осуществляют системные вызовы с помощью функций-оберток предоставляемых стандартной библиотекой языка С. Каждая такая функция-обертка реализует системный вызов в соответствии со специальной конвенцией описанной для архитектуры x86_64 и семейства операционных систем Linux в System V Application Binary Interface [4]. Согласно этой конвенции системный вызов выполняется по следующим правилам:

1. Приложения пользовательского уровня используют целочисленные регистры общего назначения для передачи аргументов системного вызова в следующей последовательности: **%rdi**, **%rsi**, **%rdx**, **%rcx**, **%r8** и **%r9**.
2. Номер системного вызова помещается в регистр **%rax**.
3. Системный вызов выполняется с помощью инструкции **syscall**.
4. Результат системного вызова возвращается в регистре **%rax** в виде целочисленного значения. Возвращаясь из системного вызова, register %rax содержит результат системного вызова. Значение в диапазоне от -4095 до -1 указывает на ошибку, код которой записывается в специальную переменную библиотеки языка С - ***errno***.

Таблица системных вызовов

Название системного вызова	Номер системного вызова	Аргументы системного вызова	Описание системного вызова
fork	57		Создаёт новый (дочерний) процесс посредством полного копирования памяти вызывающего (родительского) процесса. При успешном завершении родителю возвращается идентификатор

			дочернего процесса-потомка, а процессу-потомку возвращается 0. В случае ошибки возвращается -1.
execve	59	const char* filename, const char *argv[], const char * envp[]	Загружает в память процесса программу *filename* и запускает ее выполнение начиная с точки входа. В функцию main передается массив параметров - *argv* и в качестве третьего параметр массив параметров окружения - *envp*. При успешном выполнении execve() не возвращает управление. В случае ошибки возвращается -1.
exit	60	int error_code	Завершает выполнение текущего процесса и возвращает значение *status* родительскому процессу.
wait	247	int which, pid_t upid, struct siginfo* infop, int options, struct rusage* ru	Блокирует выполнение данного процесса для ожидания изменения состояния дочернего процесса. Для which == P_PID производится ожидание дочернего процесса с идентификатором upid. Для which == P_PGID производится ожидание любого процесса из группы процессов с идентификатором upid. Для which == P_ALL производится ожидание любого потомка, значение upid игнорируется. infop. Ожидаемая смена состояния дочернего процесса задается в options: WEXITED (полное завершение), WSTOPPED (завершение по сигналу), WCONTINUED (возобновление дочернего процесса по сигналу SIGCONT). В случае успешного выполнения wait() возвращает ID процесса завершившегося потомка. В параметре *infop* возвращается информация о состоянии

			ожидаемого дочернего процесса.
open	2	const char* filename, int flags, umode_t mode	Открывает файл *filename* в одном из следующих режимов доступа - *mode*: *O_RDONLY* (только для чтения), *O_WRONLY* (только для записи), *O_RDWR* (для чтения и записи). Также в системный вызов могут переданы флаги создания файла - flags: *O_APPEND* (открыть в режиме добавления в конец), *O_CREAT* (создать файл, если он не существует), *O_TRUNC* (опустошает существующий файл). Возвращают дескриптор открытого файла - *fd*.
close	3	unsigned int fd	Закрывает файл с дескриптором *fd*.
read	0	unsigned int fd, char* buf, size_t count	Читает *count* байт из файлового дескриптора *fd* в буфер по адресу *buf*. При успешном выполнении возвращается количество прочитанных байт (ноль означает конец файла), а позиция в файле увеличивается на это значение.
write	1	unsigned int fd, const char* buf, size_t count	Записывает *count* байт из буфера *buf* в файл, на который ссылается файловый дескриптор *fd*. В случае успеха, возвращается количество записанных байт (ноль говорит о том, что ничего записано не было).
brk	12	unsigned long addr	Устанавливает конец сегмента данных на адрес в виртуальном адресном пространстве, указанный в аргументе *addr*, если этот адрес является приемлемым, система имеет достаточно памяти и процесс не достиг максимально возможного размера своего сегмента данных. При успешном выполнении возвращает 0. В

			библиотеке языка С используется обертка над *brk(addr)* - *sbrk(increment)*, сдвигающая границу сегмента данных на *increment*, поддерживающая текущий адрес сегмента данных в глобальной переменной. Текущий адрес границы сегмента данных может быть получен с помощью системного вызова *brk(0)*.
mmap	9	unsigned long addr, unsigned long length, unsigned long prot, unsigned long flags, unsigned long fd, unsigned long offset	Создает отображение данных объекта, на который ссылается дескриптор *fd*, со смещением *offset* в адрес *addr* адресного пространства процесса размера *length* байт. Для памяти выделенной под отображение задается режим защиты *prot*: PROT_EXEC (отображенное содержимое в памяти может быть исполнено), PROT_READ (отображенное содержимое в памяти может быть прочитана), PROT_WRITE (отображенное содержимое в памяти может быть изменено), PROT_NONE (отображенное содержимое в памяти недоступно).

Упомянутые в данной таблице флаги и типы данных определяются в заголовочных файлах: sys/types.h, sys/wait.h, sys/stat.h, fnctl.h, sys/mman.h.

В языке С системный вызов может быть реализован с помощью механизма ассемблерных вставок. Например, функция системного вызова open может быть одним из следующих способов:

1. Базовый синтаксис ассемблерных вставок:

```
int open(const char *filename, int flags, int mode)
{
  register volatile int fd asm("rax");
  register int syscall_code asm("rax") = 2;

  asm("syscall");
  return fd;
}
```

2. Расширенный синтаксис ассемблерных вставок:

```
int my_open(const char *filename, int flags, int mode)
{
  int fd;

  asm("syscall"
      : "=a" (fd)
      : "a" (2), "D" (filename), "S" (flags), "d" (mode));
  return fd;
}
```

В отличии от базового синтаксиса ассемблерных вставок, который напрямую копирует переданную строку ассемблерного кода в скомпилированный компилятором ассемблерный код всей функции, расширенный позволяет передать в ассемблерный код локальные и глобальные переменные, которые будут корректно встроены компилятором в выходной ассемблерный код всей функции. Расширенный синтаксис ассемблерных вставок имеет следующую структуру:

```
asm(“АссемблерныйКод”
    : ВыходнойОперанд [ , ВыходнойОперанд ]
    : ВходнойОперанд [, ВходнойОперанд])

ВыходнойОперанд ::= “=СпецификаторРасположения” ( ЗначениеИлиПеременная )
ВходнойОперанд ::= “СпецификаторРасположения” ( ЗначениеИлиПеременная )
```

СпецификаторРасположения указывает где будет располагаться ЗначениеИлиПеременная и может принимать следующие значения:

- r - любой регистр общего назначения.
- m - память (стэк).
- D - RDI
- S - RSI
- d - RDX
- c - RCX

Для передачи параметров через регистры r8 и r9 их необходимо отобразить на соответствующие локальные переменные с помощью синтаксиса:

```
register int foo asm("r8");
```

Однако этого можно не делать в случае если прототип функции реализующей системный вызов соответствует прототипу системного вызова.

Упрощенная схема реализации отладчика

При реализации отладчика используются следующие системные вызовы и обертки над ними предоставляемые библиотекой языка C:

1. fork
2. execl
3. ptrace
4. waitpid

Системный вызов *long ptrace(int request, pid_t pid, void *addr, void *data)* принимает на вход следующие параметры в зависимости от значения параметра *request*:

1. PTRACE_TRACEME - Означает, что данный процесс будет отслеживаться его родителем.
2. PTRACE_CONT - Перезапускает остановленный отслеживаемый процесс pid. Если data не равен нулю, он интерпретируется как номер сигнала, который должен быть передан отслеживаемому процессу.
3. PTRACE_PEEKTEXT, PTRACE_PEEKDATA - Возвращает слово считанное по адресу *addr* в отслеживаемом процессе *pid*.
4. PTRACE_POKETEXT, PTRACE_POKEDATA - Копирует слово *data* по адресу *addr* отслеживаемого процесса *pid*.
5. PTRACE_GETREGS - Копирует регистры общего назначения отслеживаемого процесса pid в структуру типа *struct user_regs_struct* расположенную по адресу *data*.
6. PTRACE_KILL - Посылает отслеживаемому процессу *pid* сигнал *SIGKILL*.

Алгоритм работы программы отладчика состоит из следующих шагов [3]:

1. Отладчик создает дочерний процесс для отладки с помощью системного вызова fork(); Далее на основе значения, возвращаемого из системного вызова выполняется либо подготовка отлаживаемой программы (дочернего процесса), либо ожидание .
2. Сперва отлаживаемый процесс должен отключить механизм рандомизации адресного пространства процесса с помощью системного вызова *personality(ADDR_NO_RANDOMIZE)*. Это также можно сделать для всей операционной системы с помощью команды:

```
echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

   Дальнейшая подготовка отлаживаемого процесса состоит из двух шагов: вызов системного вызова ptrace(PTRACE_TRACEME, 0, 0, 0) для включения механизма отслеживания для текущего процесса, а после ответа от родительского процесса Отладчика в память загружается файл с отлаживаемой программой с помощью системного вызова *execl(filename, filename, NULL)*, где *filename* - путь к отлаживаемой программе.
3. Отладчик ожидает выполнения дочерним процессом системного вызова *ptrace* с помощью системного вызова waitpid, в который передается полученный от fork идентификатор дочернего процесса:

```
int status;
```

```
waitpid(child_pid, &status, 0);
```

4. После этого Отладчик устанавливает точку останова (далее Breakpoint) в отлаживаемой программе. Структура для Breakpoint может иметь следующее определение:

```
typedef struct breakpoint {
  void *instr_addr;
  long saved_code;
} breakpoint;
```

5. Для установки Breakpoint по адресу некоторой инструкции отлаживаемого процесса, данная инструкция заменяется инструкцией *int 3* (машинный код, которой: *0xcc*), которая порождает прерывание, в результате которого операционная система посылает отлаживаемому процессу сигнал *SIGTRAP*. Для замены инструкции по адресу instr_addr Отладчик сперва выполняет системный вызов *ptrace(PTRACE_PEEKTEXT, child_pid, instr_addr, NULL)*, который возвращает данные (инструкции) содержащиеся по адресу *instr_addr* в сегменте кода дочернего процесса в виде значения типа long (4 байта). Полученные данные, как и адрес инструкции, где устанавливается Breakpoint, сохраняется в структуре *breakpoint*. Затем первый байт все тех же данных устанавливается равным *0xcc*. После чего эти данные загружаются в память отлаживаемого процесса с помощью системного вызова *ptrace(PTRACE_POKETEXT, child_pid, instr_addr, modified_code)*.
6. После чего Отладчик запускает отлаживаемую программу с помощью системного вызова *ptrace(PTRACE_CONT, child_pid, 0, 0)* и ожидает от нее сигнала *SIGTRAP* снова вызывая *waitpid(child_pid, &status, 0)*.
7. Как только сигнал будет получен, Отладчик может получить содержимое регистров отлаживаемого процесса с помощью следующего кода:

```
struct user_regs_struct regs;
ptrace(PTRACE_GETREGS, child_pid, NULL, &regs);
```

8. Для снятия точки останова необходимо записать сохраненные в структуре *breakpoint* данные в память отлаживаемого процесса с помощью системного вызова *ptrace(PTRACE_POKETEXT, child_pid, b.instr_addr, b.saved_code)*.
9. Теперь отлаживаемой процесс может быть остановлен с помощью системного вызова *ptrace(PTRACE_KILL, child_pid, 0, 0)*.

Задания

1. Реализовать системные вызовы в соответствии с вариантом деленному по модулю 4. При выполнении данного задания можно

использовать ассемблерные вставки, синтаксис которых для компилятора GCC описан в [2]. Список системных вызовов для реализации:

1. fork, open, mmap;
2. execve, close, mmap;
3. exit, read, mmap;
4. wait, write, brk.

Проверить работоспособность реализованных системный вызовов.

2. С использованием описанных в теоретической части системных вызовов, реализовать программу Отладчик. Данная программа должна выводить содержимое регистров общего назначения центрального процессора в момент, когда выполнение отлаживаемой программы доходит до некоторого символа (функции), то есть когда в регистр RIP загружен адрес этого символа (функции) в памяти процесса. Программа должна принимать на вход путь к исполняемому файлу отлаживаемой программы и адрес символа в файле отлаживаемой программы. Таблица адресов символов некоторой программы может быть получена с помощью команды *nm путь/к/программе.*

Замечание 1: при выполнении данного задания сперва необходимо отключить рандомизацию адресного пространства процессов.

Замечание 2: при подмене инструкций отлаживаемой программы необходимо использовать адрес символа в адресном пространстве процесса. Для этого к несмещенному адресу символа в файле программы (полученному с помощью утилиты *nm*) нужно добавить базовый адрес сегмента кода в памяти процесса, который можно получить из карты адресного пространства выводимой командой:
cat /proc/self/maps

3. По выполненной работе нужно оформить отчет, содержащий описание ключевых аспектов реализации и демонстрацию работы написанных программ.

Ссылки

1. Searchable Linux Syscall Table for x86 and x86_64: https://filippo.io/linux-syscall-table/
2. Extended Asm - Assembler Instructions with C Expression Operands: https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html#Extended-Asm
3. Writing a Linux Debugger Part 2: Breakpoints: https://blog.tartanllama.xyz/writing-a-linux-debugger-breakpoints/
4. System V Application Binary Interface: https://www.uclibc.org/docs/psABI-x86_64.pdf