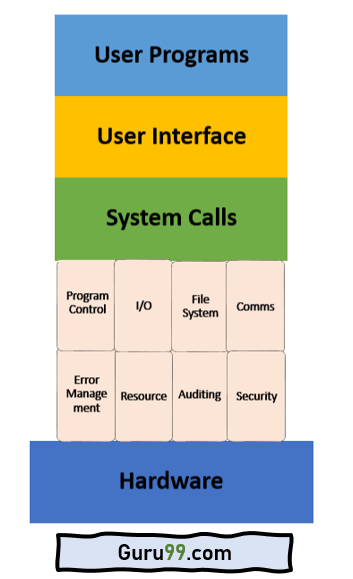
**Лекция 4**

**fun fact:** *uname -a* — посмотреть версию убунты

В этот раз поговорим про системные вызовы.

**Системный вызов** — это функция, которая осуществляет взаимодействие между пользователем и операционной системой. Системный вызов - интерфейс, предоставляемый ОС программам, прослойка между ними и пользователем.



Посмотрим, какие есть системные вызовы.

**man** — команда, чтобы посмотреть документацию какой-то программы.

Во 2 разделе документации находятся системные вызовы (например, введя *man 2 write* — получим описание сискола write)

*man syscalls* — выводит список сисколов

**strace** — показывает, какие системные вызовы делает программа. *strace ./a.out*

**ltrace** — позволяет следить за тем, какие библиотечные вызовы делает программа.

Выведем с помощью сисколов текст на консоль:

<unistd.h> - в этом файле объявлены некоторые сисколы

Посмотрим, что нужно писать во write: ssize\_t write(int fd, const void \*buf, size\_t count);

fd — файловый дескриптор — число, которое является идентификатором того, куда хотим писать.

* 0 — стандартный поток ввода
* 1 — стандартный поток вывода
* 2 — стандартный поток ошибок

#include <unistd.h>  
  
int main() {  
 const char \*str = "Hello world!";  
 write(1, str, 12);  
}

Но такой способ, на самом деле, небезопасный. ОС не гарантирует, что сисколы отработают без ошибок. Могла случиться проблема, и в итоге записалось бы меньше символов, чем мы просили, или не записать ничего.

#include <unistd.h>  
#include <cstring>  
#include <errno.h>  
  
int main() {  
 const char \*str = "Hello world!";  
 long i = write(3, str, strlen(str));  
 if (i == -1) {  
 write(2, "fail!", 5);  
 } else {  
 while (strlen(str) - i > 0) {  
 i += write(1, str, strlen(str) - i);  
 }  
 }  
}

Вот так, кажется, более правильно, но хз, запись будет - поправлю.

Ввод делается с помощью сискола read - читает из файлового дескриптора в буфер (в душе не знаю, как это правильно писать, ну пофиг потом перепишу).

#include <unistd.h>  
  
int main() {  
 char buf[100];  
 read(0, buf, 10);  
 write(2, "hello world ", 12);  
 write(2, buf, 10);  
 \_exit(0);  
}

Если в итоге выполнения мы увидили fail!, можно узнать, по какой причине это случилось. Для этого существует **errno**. Это некоторая глобальная переменная, в которую записывается код последней случившейся ошибки.

*errno -l* — посмотреть список ошибок.

#include <unistd.h>  
#include <cstring>  
#include <errno.h>  
#include <string>  
  
int main() {  
 const char \*str = "Hello world!";  
 long i = write(3, str, strlen(str));  
 if (i == -1) {  
 write(2, "fail!", 5);  
 std::string err = std::to\_string(errno);  
 write(1, err.data(), err.size());  
 } else {  
 while (strlen(str) - i > 0) {  
 i += write(1, str, strlen(str) - i);  
 }  
 }  
}

Ну вот так этим пользоваться ±. Выведется fail!9 — код ошибки означает, что пишем в неправильный файловый дескриптор. У errno также может быть значение глобальной константы EBADF (bad file descriptor)

Если попробовать вывести в cout 1000 символов, и посмотреть через strace, сколько вызовов write делается, увидим, что запись делается порционно. Когда мы что-то пишет в cout, он не сразу выводит данные на консоль. У него есть внутренний массив, в котором он хранит данные и копит их туда, а потом при переполнении выводит (когда накопится 1024 символа). Это делается для того, чтобы реже вызывать сисколы, потому что это довольно дорогая операция. Именно поэтому, когда случайно в программе появляется segfault, мы можем не увидеть то, что было записано в cout, потому что буфер мог не успеть очиститься. Поэтому мы используем **std::cout.flush() -**  заставляет cout вывести буфер.

#include <iostream>  
#include <unistd.h>  
  
  
int main() {  
 for (int i=0; i < 1000; ++i) {  
 std::cout << i;  
 }  
 \_exit(0);  
}

Если дописать \_exit(0), то выведутся не все числа. Если просто exit(0), то норм. Все потому, что **exit** - библиотечная функция, делающая системный вызов (и еще много других сисколов, которые нужно сделать при выходе из программы), а **\_exit** — syscall, с помощью него мы обращаемся к системе напрямую. И с помощью него мы обрубаем выполнение программы, не дожидаясь, пока cout выведет значения из буфера.

**open** — syscall, открывает файл.

**creat** — функция, которая позволяет создать файл.

Системный вызов open (*man 2 open*, там же можно посмотреть нужные библиотеки) возвращает id, по которому в дальнейшем в ОС мы будем идентифицировать открытый файл.

У open также можно задать флаг — глобальную целочисленную константу (man 2 open — список флагов)

#include <string>  
#include <unistd.h>  
#include <fcntl.h>  
  
int main() {  
 int fd = open("/home/marypan/input.txt", O\_RDONLY);  
 write(1, std::to\_string(fd).data(), 1); // выведет id, например, тройку  
}

#include <string>  
#include <unistd.h>  
#include <fcntl.h>  
  
int main() {  
 int fd = open("/home/marypan/input.txt", O\_RDONLY);  
 char buff[1000];  
 read(fd, buff, 10);  
 write(1, buff, 10);  
}

Что если читать из файла, который изменится или будет удалён другой программой? При удалении мы, скорее всего, будем ловить ошибки. При изменении, скорее всего, будет UB на уровне syscall'ов.

open не создает файл, если его не существовало. Но если добавить флаг O\_CREAT, то файл создастся, если его не было.

Еще по-хорошему надо вызывать close(file), чтобы все было хорошо… (иначе мб UB?)

Примечание

**ps** - показать процессы. Например, ps -ef | grep firefox.

**g++ -no-pie test.cpp** - no position, independent, executable. Формат pie был придуман для защиты от хакерских атак. Однако для readelf такие файлы могут считаться динамическими (DYN).