**Лекция 10**

**Как подменить malloc на свой собственный?**

Это можно сделать благодаря тому, что malloc — слабый символ. И если с помощью переменной LD\_PRELOAD в параметрах запуска программы передать адрес другой библиотеки, в которой находится переопределенный маллок, то так мы сможем заменить стандартный маллок.

Если переопределяем malloc, то обязаны также доопределить функции free, calloc и realloc, иначе UB.

**Процессы**

Что происходит в ОС, когда она работает? Что такое вообще сисколы? Как вообще работает процессор?

Есть процессор, и у него есть последовательность инструкций, **instruction pointer**, и он выполняет команды, которые записаны в бинарном коде, такого типа: положи в ячейку число, сложи две ячейки, вызови функцию, прочитай из памяти, запиши в память и т.д. Но в какой-то момент приходит другой процесс. На одном ядре могут выполняться разные процессы. Как происходит снятие процесса и выполнение нового? Для этого существует механизм прерываний — **interrupts**.

Один из примеров прерываний — таймер, который есть у процесса. Как процесс получает доступ к процессору? ОС, прежде чем поставить что-то на исполнение, заводит таймер. У процессора есть также обработчики прерываний — **interrupt handlers**, загруженные заранее ОС. В памяти процессора есть некая таблица вида событие — действие. Поэтому если произошло какое-то событие, тикнул таймер к примеру, процессор знает, какое действие ему нужно совершить: пойти исполнять код на какой-то адрес, например. А на этом адресе уже лежит код ОС, который начинает порядки наводить. Посл завершения своих действий снова заводится таймер, снова процессор идет исполнять код процесса. Из своего кода хендлеры поменять нельзя, это можно сделать только из привилегированного режима, в котором работает только код ядра ОС. Pagefault — также пример прерывания.

**Запускаем из кода новый процесс**

**top** — стандартная команда, которая показывает таблицу работающих процессов, обновляющуюся итеративно.

У каждого процесса есть свой айди — pid, user — кто его запустил, приоритет pr, и число, противоположное приоритету — ni (niceness).

**fork** — сискол, который запускает дочерний процесс.

*pid\_t fork(void)*

Заходим в fork, выходим, а нас двое стало!

А что такое сискол вообще? До сих пор мы воспринимали это как вызов функции из операционной системы. На самом деле любой сискол — это функция, которая работает так: при выполнении ее, наш процесс снимается с выполнения и начинает выполняться код ОС, а потом процесс возвращается к выполнению.

У всех сисколов есть номера, поэтому процессор запоминает в специальном регистре, какой сискол был вызван, останавливает выполняемый процесс, и прыгает на соответствующий номеру сискола хендлер. И никто не гарантирует возвращение оттуда, вообще говоря.

Когда вызываем fork, ОС создает почти полную копию процесса (с разными pid), и в какой-то момент возвращает управление первому процессу, и также второму.

Как понять, какой сейчас мы процесс? Тому процессу, который вызывал fork, вернется pid нового процесса, а склонированному процессу вернется 0. Если вернулось -1, процесс создать не удалось.

**Пример:**

#include <iostream>  
#include <unistd.h>  
  
int main() {  
 int pid = fork();  
 if (pid == 0) {  
 std::cout << "I'm a child! (I was born)" << '\n';  
 } else {  
 std::cout << "I am parent" << '\n';  
 }  
}

**Вопрос на удос:**

#include <iostream>  
#include <unistd.h>  
  
int main() {  
 std::cout << "Hello";  
 int pid = fork();  
 if (pid == 0) {  
 std::cout << "I'm a child! (I was born)" << '\n';  
 } else {  
 std::cout << "I am parent" << '\n';  
 }  
}

Hello выведется два раза, потому что все полностью склонируется, вся память, а значит и буфер cout. Если его очистить до создания процесса, то выведется только один раз.

**Fork bomb**

#include <unistd.h>  
int main() {  
 while (true) {  
 fork();  
 }  
}

:(){ :|:& };: - еще вот это форк бомба))

**execve** — сискол: засыпаем и пишем «хочу проснуться в теле...». Заходим в сискол, и выполнение нашей проги заканчивается, и начинает выполняться другая, но с прошлым pid.

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

**which** команда — узнать по какому пути лежит команда

**Пример использования execve:**

#include <unistd.h>  
#include <cstdio>  
  
int main() {  
 int pid = fork();  
 if (pid == 0) {  
 char\* argv[] = {"/usr/bin/ls", "directory", NULL};  
 char\* envp[] = {NULL};  
 printf("I was born\n");  
 execve("/usr/bin/ls", argv, envp);  
 printf("After execve\n");  
 } else {  
 printf("I'm parent\n");  
 }  
}

Примечание: файлы будут выведены с новой строки, потому что родитель завершился. буфер execve, кажется, почистит, но деструкторы вызывать не будет.

Над функцией execve есть много оберток.

Посмотреть их можно в man execv. Семейство этих функций exec() заменяет текущий образ процесса новым образом процесса.

По-хорошему, родители должны дожидаться, пока их ребенок завершит работу.

Каждый процесс кроме pid имеет **ppid** — id процесса, который породил этот процесс. Если ребенок завершается раньше родителя, ОС не удаляет информацию о ребенке, потому что родитель может ее запросить. Он может вызвать специальную функцию, которая позволяет дождаться завершения ребенка и получить о нем некоторую информацию — функция **wait**.

*pid\_t wait(int\* wstatus)*

В wstatus записывается результат завершения ребенка. Возвращает id завершившегося ребенка.

**sleep** — обертка над сисколом, позволяет заснуть на некоторое количество секунд.

*unsingned int sleep(unsigned int seconds)*