**Лекция 12**

**IPC** — Inter Process Communication

*man capabilities* — посмотреть список привилегий процессов

**Пример:**

#include <iostream>  
#include <sys/capability.h>  
#include <unistd.h>  
#include <sys/types.h>  
  
void setCapabilities() {  
 // Define header and data structures  
 struct \_\_user\_cap\_header\_struct header;  
 struct \_\_user\_cap\_data\_struct data[2]; // 2 для поддержки всех capabilities (64 бита)  
  
 // Initialize the header  
 header.version = \_LINUX\_CAPABILITY\_VERSION\_3;  
 header.pid = 0; // 0 означает текущий процесс  
  
 // Get current capabilities  
 if (capget(&header, data) == -1) {  
 perror("capget");  
 return;  
 }  
  
 // Modify capabilities  
 data[CAP\_NET\_ADMIN / 32].effective |= (1 << (CAP\_NET\_ADMIN % 32));  
 data[CAP\_NET\_ADMIN / 32].permitted |= (1 << (CAP\_NET\_ADMIN % 32));  
 data[CAP\_NET\_ADMIN / 32].inheritable |= (1 << (CAP\_NET\_ADMIN % 32));  
  
 // Apply the modified capabilities  
 if (capset(&header, data) == -1) {  
 perror("capset");  
 return;  
 }  
  
 std::cout << "Capabilities set successfully" << std::endl;  
}  
  
int main() {  
 setCapabilities();  
}

Здесь используется массив data для представления 64-битных capability.

Добавлена работа с битами: используются побитовые операции с учётом границ 32-битных слов.

**Еще один вариант:**

#include <iostream>  
#include <sys/capability.h>  
#include <unistd.h>  
  
void setCapabilities() {  
 // Получение текущих capability  
 cap\_t caps = cap\_get\_proc();  
 if (!caps) {  
 perror("cap\_get\_proc");  
 return;  
 }  
  
 // Установить необходимые capability  
 cap\_value\_t capList[] = {CAP\_NET\_ADMIN}; // Список capabilities  
 if (cap\_set\_flag(caps, CAP\_EFFECTIVE, 1, capList, CAP\_SET) == -1) {  
 perror("cap\_set\_flag (EFFECTIVE)");  
 cap\_free(caps);  
 return;  
 }  
 if (cap\_set\_flag(caps, CAP\_PERMITTED, 1, capList, CAP\_SET) == -1) {  
 perror("cap\_set\_flag (PERMITTED)");  
 cap\_free(caps);  
 return;  
 }  
 if (cap\_set\_flag(caps, CAP\_INHERITABLE, 1, capList, CAP\_SET) == -1) {  
 perror("cap\_set\_flag (INHERITABLE)");  
 cap\_free(caps);  
 return;  
 }  
  
 // Применение capability  
 if (cap\_set\_proc(caps) == -1) {  
 perror("cap\_set\_proc");  
 cap\_free(caps);  
 return;  
 }  
  
 std::cout << "Capabilities set successfully" << std::endl;  
  
 // Освобождение памяти  
 cap\_free(caps);  
}  
  
int main() {  
 setCapabilities();  
}

Компилируем с флагом **-lcap**

Будучи рутом, можно запустить какую-то программу, которая выдаст привилегии другой программе.

*man sched* — посмотреть набор сисколов, которые позволяют управлять планировщиком, тем, как он расставляет процессы. Он запускает на выполнение процессы по очереди: ядро системы ставит на ядро процессора какой-то процесс, и ставит таймер. Когда таймер тикает, запускается обработчик процессов, начинает исполняться код ОС. ОС решает, какой процесс поставить следующим, опять заводит таймер, и т.д.

Прикольный сискол: setaffinity (getaffinity) — установить, на каких ядрах процессу исполняться.

**Пример:**

#include <sched.h>  
#include <unistd.h>  
#include <iostream>  
  
void setCpuAffinity(int cpu) {  
 cpu\_set\_t cpuset; // bitmask  
  
 // clear the CPU set  
 CPU\_ZERO(&cpuset);  
  
 // add the specified CPU to the set   
 CPU\_SET(cpu, &cpuset);  
  
 // set the CPU affinity for the calling thread  
 if (sched\_setaffinity(getpid(), sizeof(cpu\_set\_t), &cpuset) == -1) {  
 perror("sched\_setaffinity");  
 return;  
 }  
  
 std::cout << "CPU affinity set to CPU " << cpu << std::endl;  
}  
  
int main() {  
 setCpuAffinity(0);  
 while (true) {  
 // stimulate work  
 }  
}

*man seccomp* — запретить что-то делать (вызвать сискол например). Он навешивает фильтр на те сисколы, которые можно выполнять.

**Пример:**

#include <seccomp.h>  
#include <sys/prctl.h>  
#include <unistd.h>  
#include <iostream>  
  
void setup\_seccomp() {  
 // create a new seccomp filter  
 scmp\_filter\_ctx ctx = seccomp\_init(SCMP\_ACT\_ALLOW); // default action is to allow  
  
 if (ctx == nullptr) {  
 perror("seccomp\_init");  
 exit(1);  
 }  
  
 // add rules to deny specific syscalls  
 // context, action, syscall, addition flags  
 seccomp\_rule\_add(ctx, SCMP\_ACT\_ERRNO(EPERM), SCMP\_SYS(execve), 0);  
 seccomp\_rule\_add(ctx, SCMP\_ACT\_ERRNO(EPERM), SCMP\_SYS(fork), 0);  
  
 // load the filter into the kernel  
 if (seccomp\_load(ctx) < 0) {  
 perror("seccomp\_load");  
 exit(1);  
 }  
  
 // release the filter context  
 seccomp\_release(ctx);  
}  
  
int main() {  
 setup\_seccomp(); // (без этого просто ls запустился бы)  
  
 std::cout << "Process is running with restricted syscalls " << std::endl;  
  
 char \*args[] = {"/bin/ls", NULL};  
 execve(args[0], args, NULL); // will fail  
  
 while (true) {}  
}

Если в качестве action поставить SCMP\_ACT\_KILL\_PROCESS получим Bad system call

**Какие бывают виды IPC?**

Осуществляется с помощью:

1. сигналов

2. pipes and fifo

3. разделяемой памяти

Первый механизм межпроцессного взаимодействия — **сигналы**.

*man 7 signal* — список сигналов

Есть возможность попросить ОС послать сигнал процессу (себе или постороннему, если достаточно прав).

**Сигнал** — событие, с помощью которого ОС может оповестить процесс о том, что что-то произошло (не содержит информации, от какого процесса он пришел).

**Действия сигналов:**

* Term Завершить процесс
* Ign Игнорировать процесс
* Core Завершить процесс и сделать core dump
* Stop Остановить процесс
* Cont Возобновить процесс

ctrl C — SIGINT

ctrl \ — SIGQUIT

ctrl Z — SIGTSTP

SIGKILL отличается от SIGTERM тем, что его нельзя перехватить (некоторые процессы могут блокировать сигналы или как-то реагировать в ответ на них). Таким же свойством обладает и SIGSTOP.

SIGUSR1

SIGUSR2 — пользовательские сигналы

SIGCHLD — сигнал, который посылается родителю в момент, когда завершился его ребенок.

kill — команда, которая посылает сигнал процессу. Есть одноименный сискол.

С помощью нее можно сымитировать любую ситуацию, будь то segfault или sigill.

*kill -11 $(pgrep a.out)*

Текст результата в консоль выводит дефолтный обработчик сигнала, он же завершает программу (можно и свой написать).

**Кастомный обработчик**

**signal** — сискол, который позволяет установить кастомный обработчик сигналов. Устаревший способ.

*sighandler\_t signal(int signum, sighanddler\_t handler);*

Пример:

#include <stdio.h>  
#include <signal.h>  
  
void handler(int signum) {  
 printf("Signal number %d received\n", signum);  
}  
  
int main() {  
 signal(SIGINT, &handler);  
 while (true) {}  
}

SIGKILL заблокировать нельзя.

Что будет?

#include <stdio.h>  
#include <signal.h>  
  
void handler(int signum) {  
 printf("Signal number %d received\n", signum);  
}  
  
int\* p = NULL;  
  
int main() {  
 signal(SIGSEGV, &handler);  
 \*p = 5;  
}

бесконечно будет писать надпись сигнал получен. Когда обработка сигнала заканчивается, мы возвращаемся в то место, откуда пришел сигнал. То есть пытаемся повторить ту же самую инструкцию.

А вот так?

#include <stdio.h>  
#include <signal.h>  
  
int\* p = NULL;  
int a = 0;  
  
void handler(int signum) {  
 printf("Signal number %d received\n", signum);  
 p = &a;  
}  
  
int main() {  
 signal(SIGSEGV, &handler);  
 \*p = 5;  
}

Исход такой же будет. Все потому, что после обработки сигнала мы возвращаемся не на ту же строку кода, а на ту же самую ассемблерную инструкцию. А после изменения p адрес того регистра процессора, куда мы пытались записать 5, не поменялся.

Минутка вспомнить:

Volatile в C++ — ключевое слово, которое информирует компилятор о том, что значение переменной может меняться извне (например, под управлением операционной системы, аппаратных средств или другого потока).

Это предотвращает компилятор от некоторых оптимизаций, предполагая, что значение переменной остаётся неизменным, если не изменено явно в программе.

Тогда вот так:

#include <stdio.h>  
#include <signal.h>  
#include <sys/mman.h>  
  
int\* p;  
  
void handler(int signum) {  
 printf("Signal number %d received\n", signum);  
 mprotect(p, 1000, PROT\_READ | PROT\_WRITE);  
}  
  
int main() {  
 signal(SIGSEGV, &handler);  
  
 p = (int\*)mmap(0, 1000, PROT\_READ, MAP\_PRIVATE | MAP\_ANONYMOUS, -1, 0);  
 \*p = 5;  
  
 printf("%d", \*p);  
 munmap(p, 1000);  
}

Так уже работает.

На самом деле все что выше это UB, потому что в обработчике сигналов нельзя вызывать printf, как и многие другие функции, которые поддерживают буфер. Процесс может быть снят с выполнения в любой момент, в том числе как раз посреди выполнения printf. Потом пришел сигнал, в котором мы снова вызываем printf — на той же памяти, при том, что предыдущий printf еще не завершился.

*man signal-safety* — список функций, которые могут быть безопасно вызваны из обработчика сигналов. Вызов остальных функций — UB.

А что, если поймали сигнал находясь в обработчике сигналов? Придется начать обрабатывать и его!

Обработчики сигналов не должны быть тяжеловесными!

**Пример:**

#include <stdio.h>  
#include <signal.h>  
  
int\* p;  
  
void handler(int signum) {  
 printf("Signal number %d received\n", signum);  
 while (true) {}  
}  
  
int main() {  
 signal(SIGQUIT, &handler);  
 signal(SIGINT, &handler);  
 while (true) {}  
}

Если нажмем первый раз ctrl C и ctrl \, то обработчики сигналов выведут надпись, но при дальнейших нажатиях ничего не произойдет, потому что находясь в обработчике сигнала, мы можем поймать все сигналы, кроме тех, что уже обрабатываем.

ОС запоминает, в каких обработчиках мы сейчас находимся и какие сигналы сейчас не можем посылать. Если придет запрещенный сигнал, ОС запомнит его и когда выйдем из обработчика, сможет его послать.

**Пример для демонстрации:**

#include <stdio.h>  
#include <signal.h>  
  
int\* p;  
  
void handler(int signum) {  
 printf("Signal number %d received\n", signum);  
 sleep(5);  
}  
  
int main() {  
 signal(SIGQUIT, &handler);  
 signal(SIGINT, &handler);  
 while (true) {}  
}

Но количество сигналов не запоминается:

В Unix-системах сигналы не ставятся в очередь, если они одного типа. Когда сигнал SIGINT обрабатывается, любые дополнительные сигналы того же типа (SIGINT) игнорируются до завершения обработчика.

Играться с этим можно с помощью struct sigaction.