**Лекция 13**

Что случится, если во время обработки segfault словили еще segfault?

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

На такой случай есть защита: второй раз этот обработчик уже не вызовется, а вызовется стандартный обработчик.

А вообще допущение сегфолта и подобного в обработчике сигналов это UB.

**kill** — послать сигнал процессу. Либо процесс должен быть вашим, либо вы должны иметь права на это.

**raise** — функция, позволяющая послать сигнал самому себе.

Пример функции, которая посылает сигнал самой себе: **abort**

На SIGABRT можно поставить обработчик.

**pause** — сискол, вызов которого приводит к остановке программы до тех пор, пока не придет сигнал.

**SIGHUP** — сигнал, такой что:

через SSH зашли на удаленный сервер, и хотим выполнить на нем какое-то вычисление. Оно работает продолжительное время, а в течение этого времени сеть отвалилась. В этот момент процессу на сервере придет сигнал SIGHUP и он останавливается (трубку положили).

**nohup** — команда, которая устанавливает обработчик на SIGHUP, который игнорирует его, и процесс продолжается.

*nohup ./a.out*

Другой способ обработки сигналов — **sigaction**.

*int sigaction(int signum, const struct sigaction \*act, struct sigaction \*oldact);*

**Пример:**

#include <stdio.h>  
#include <signal.h>  
#include <stdlib.h>  
  
void handler(int signum) {  
 printf("Signal number %d received\n", signum);  
 exit(1);  
}  
  
int main() {  
 struct sigaction sa;  
 sa.sa\_handler = &handler;  
 sigemptyset(&sa.sa\_mask); // занулить биты маски  
 sa.sa\_flags = 0;  
  
 if (sigaction(SIGSEGV, &sa, NULL) == -1) {  
 perror("sigaction");  
 exit(1);  
 }  
  
 int\* ptr = NULL;  
 \*ptr = 10;  
  
 printf("Continuing execution after segfault\n");  
}

struct sigaction {

void (\*sa\_handler)(int); // функция, которая будет вызвана

void (\*sa\_sigaction)(int, siginfo\_t \*, void \*); // другой вид обработчика

sigset\_t sa\_mask; // отвечает за то, какие сигналы блокируем

int sa\_flags; // указать дополнительные флаги

void (\*sa\_restorer)(void);

};

sigaction дает возможность не блокировать сигналы в момент, когда обрабатываются другие сигналы.

Если прервали sleep сигналом, то по возвращении из обработчика обратно в sleep не возвращаемся (таймер не продолжаем учитывать).

**SA\_ONSTACK** — попросить обработчик обрабатывать сигналы на новом стеке. Если другой стек не получается использовать, используется дефолтный.

**sigaltstack** — позволяет потоку определять новый альтернативный стек сигналов

и/или извлекать состояние существующего альтернативного стека сигналов.

Альтернативный стек сигналов используется во время выполнения обработчика сигналов, если его запросило создание этого обработчика.

**sigpending** — функция, которая показывает, какие сигналы должны были быть отправлены, но еще не отправились.

**sigprocmask** — посмотреть заблокированные сигналы.

**Pipes — второй вид IPC**

Позволяют передавать некоторый поток информации от одного процесса другому.

**pipe** — сискол, который позволяет создать пайп (однонаправленный канал передачи данных, который может использоваться для межпроцессного взаимодействия).

*int pipe(int pipefd[2]);*

Когда создаем пайп, в ответ ОС выдает 2 файловых дескриптора, поэтому в этот сискол мы передаем массив из двух интов.

pipefd[0] — конец трубы, откуда можно читать

pipefd[1] — конец трубы, в который можно писать

Данные, которые записываются в pipe с конца откуда запись, буферизуются ядром, до тех пор, пока не будут прочитаны с другого конца.

В рамках одного дерева процессов с помощью пайпов процессы могут взаимодействовать между собой. То есть процесс может общаться либо с ребенком, либо его дети могут общаться между собой. Когда делаем fork, файловые дескрипторы клонируются.

**Пример:**

#include <stdio.h>  
#include <unistd.h>  
#include <errno.h>  
#include <sys/wait.h>  
#include <string.h>  
#include <stdlib.h>  
  
int main(int argc, char\* argv[]) {  
 char buf[1000];  
 int pipefd[2];  
 pipe(pipefd);  
  
 int cpid = fork();  
 if (cpid == 0) { // child reads from pipe  
 close(pipefd[1]); // close unused write end  
  
 while (read(pipefd[0], &buf, 1) > 0) {  
 write(1, &buf, 1);  
 }  
 write(1, "\n", 1);  
 close(pipefd[0]);  
 \_exit(0);  
 } else { // parent writes argv[1] to pipe  
 close(pipefd[0]); // close unused read end  
 sleep(1);  
 write(pipefd[1], argv[1], strlen(argv[1]));  
 close(pipefd[1]);  
 wait(NULL); // wait for child  
 exit(0);  
 }  
}

Что в этом примере происходит: родитель сначала спит секунду, потом пишет. А ребенок читает сразу, пока pipe еще пустой. Когда вызываем read на пустой буфер, мы блокируемся (поэтому без close[pipefd[1]) в родителе программа не завершалась), и из read не возвращаемся, пока в pipe не запишут что-то. Это называется блокирующее чтение. По итогу чтобы выйти из read наконец (потому что он будет ждать еще что-то в пайпе), нужно закрыть пайп.

А что, если писать в пайп, который никто не читает? Write тоже заблокируется.

**SIGPIPE** — пишем в пайп, который никто не читает.

#include <stdio.h>  
#include <unistd.h>  
#include <errno.h>  
#include <sys/wait.h>  
#include <string.h>  
#include <stdlib.h>  
  
void handler(int signum) {  
 printf("Catch SIGPIPE\n");  
 \_exit(0);  
}  
  
int main(int argc, char\* argv[]) {  
 signal(SIGPIPE, &handler);  
 char buf[1000];  
 int pipefd[2];  
 pipe(pipefd);  
  
 int cpid = fork();  
 if (cpid == 0) { // child reads from pipe  
 close(pipefd[1]); // close unused write end  
 close(pipefd[0]);  
 /\*while (read(pipefd[0], &buf, 1) > 0) {  
 write(1, &buf, 1);  
 }\*/  
 sleep(5);  
 write(1, "\n", 1);  
 close(pipefd[0]);  
 \_exit(0);  
 } else { // parent writes argv[1] to pipe  
 close(pipefd[0]); // close unused read end  
 sleep(1);  
 write(pipefd[1], argv[1], strlen(argv[1]));  
 close(pipefd[1]);  
 wait(NULL); // wait for child  
 exit(0);  
 }  
}

В этом примере SIGPIPE ловится. По сути что происходит: write ловит SIGPIPE, который делает terminate родителя, а ребенок еще живет. Он через 5 секунд пишет перевод строки и завершается.

Подобным образом можно общаться и между произвольными процессами. Для этого нужны именованные пайпы — fifo файлы.

**mkfifo** — создать фифо файл.

Фифо не хранится на диске, это абстракция. В фифо также можно писать и читать. Аналогично пайпу все.

Такая же функция для создания фифо из кода есть в сишной библиотеке — mkfifo.

*int mkfifo(const char\* pathname, mode\_t mode)*

**Пример использования:**

#include <sys/stat.h>  
#include <unistd.h>  
#include <string>  
#include <string.h>  
#include <fcntl.h>  
  
int main() {  
 mkfifo("./test\_fifo", 0666);  
 int fd = open("./test\_fifo", O\_WRONLY);  
  
 char\* str = "Hello\n";  
 int count = 0;  
 while (true) {  
 ++count;  
 write(fd, str, strlen(str));  
 std::string countstr = std::to\_string(count) + "\n";  
 write(1, countstr.data(), strlen(countstr.data()));  
 }  
}

Когда начнем читать cat test\_fifo, а затем прервем чтение, то завершится и сам процесс. Пока никто не читал, он висел и писал кучу всего.

Оператор | - это пайп. Установили трубу между двумя процессами:

ls | grep mmap

И получаем таким образом, что один процесс пишет в другого.

Оператор && конъюнкция. Выполняет вторую часть, только если первая завершилась удачно.

Оператор || дизъюнкция наоборот, вторая часть выполняется, если первая завершилась неудачно.