1. Разработать приложение, реализующее обработку сигналов HUP, INT, ABRT с помощью функции signal. Приложение должно информировать о получении соответствующего сигнала вне обработчика. Продемонстрировать работу скрипта вызовом команды kill.
2. Разработать приложение, реализующее обработку сигналов HUP, INT, ABRT с помощью сигнальной маски. Продемонстрировать работу скрипта вызовом команды kill.
3. Реализовать безопасную обработку сигналов HUP, INT, ABRT с помощью системного вызова sigaction. Продемонстрировать работу скрипта вызовом команды kill.
4. Разработать многопоточное приложение, состоящее из 4 потоков, где каждый поток обрабатывает только один сигнал: первый поток - HUP, второй - INT, третий - ABRT, главный поток - порождает дочерние потоки. Для каждого потока должен быть реализован отдельный класс.

# Приложение, реализующее обработку сигналов HUP, INT, ABRT с помощью функции signal

В рамках выполнения задания был изучен порядок обработки сигналов с помощью функции signal() в среде CLion OC Linux. Сигнал - программное прерывание процесса. Операционная система использует сигналы, чтобы сообщить исключительные ситуации выполняемой программе. Вызов signal() устанавливает обработчик сигнала с номером signum в соответствии с параметром handler, который может быть равен SIG\_IGN, SIG\_DFL или адресу функции пользователя ("обработчик сигнала").

Листинг 1. Приложение, реализующее обработку сигналов HUP, INT, ABRT

#include <signal.h>

#include <unistd.h>

#include <stdio.h>

#include <string.h>

volatile sig\_atomic\_t signal\_number, counter = 5;

void \*handler(int signum) {

signal(signum, SIG\_IGN);

// запоминаем номер сигнала, не тратим время на вывод

signal\_number = signum;

// восстанавливаем обработчик по умолчанию для завершения

signal(signum, counter ? handler : SIG\_DFL);

// во время обработки может прийти другой сигнал!!!

}

int main() {

signal(SIGINT, handler);

signal(SIGHUP, handler);

signal(SIGABRT, handler);

while(counter--) {

pause();

// вывод производим вне обработчика

printf("signal: %s\n", strsignal(signal\_number));

}

}

Результат выполнения листинга 1 приведен на рисунке 1.

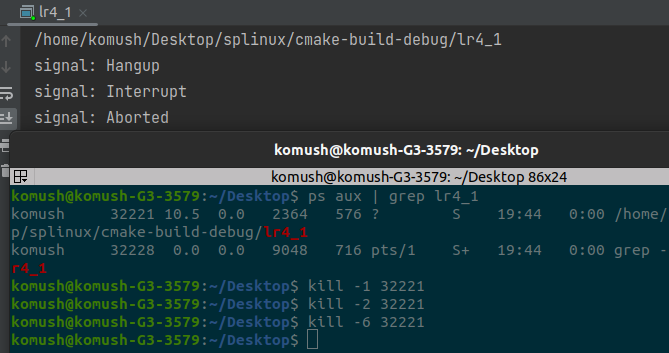


Рисунок 1. – Результат работы приложения

# Приложение, реализующее обработку сигналов HUP, INT, ABRT с помощью сигнальной маски

В рамках разработки приложения был изучен порядок работы с сигнальной маской, с помощью которой обрабатываюсь сигналы HUP, INT, ABRT. В каждой нити процесса имеется независимая сигнальная маска, определяющая набор сигналов, которые нить, в данный момент, блокирует. Нить может управлять сигнальной маской с помощью [pthread\_sigmask](https://ru.manpages.org/pthread_sigmask/3). В обычном однонитиевом приложении для работы с сигнальной маской можно использовать вызов [sigprocmask](https://ru.manpages.org/sigprocmask/2).

Листинг 2. Приложение, реализующее обработку сигналов HUP, INT, ABRT.

#include <signal.h>

#include <unistd.h>

#include <stdio.h>

#include <string.h>

volatile sig\_atomic\_t signal\_number, counter = 5;

void \*handler(int signum) {

sigset\_t sigset, oldset;

sigfillset(&sigset);

sigprocmask(SIG\_SETMASK, &sigset, &oldset);

signal\_number = signum;

sigprocmask(SIG\_SETMASK, &oldset, &sigset);

}

int main() {

signal(SIGINT, handler);

signal(SIGHUP, handler);

signal(SIGABRT, handler);

while(counter--) {

pause();

printf("signal: %s\n", strsignal(signal\_number));

}

}

Результат выполнения листинга 2 приведен на рисунке 2.

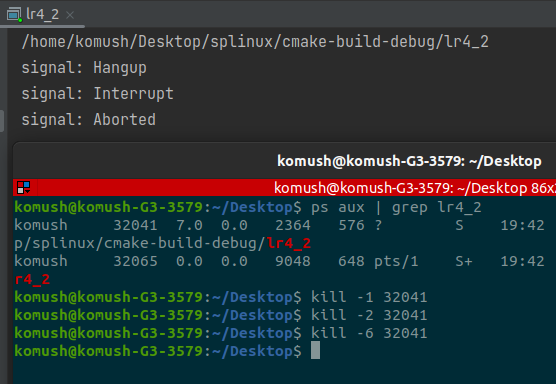


Рисунок 2. – Результат работы приложения

# Безопасная обработку сигналов HUP, INT, ABRT с помощью системного вызова sigaction

В рамках выполнения задания был изучен порядок обработки сигналов HUP, INT, ABRT в среде CLion OC Linux. Системный вызов sigaction используется для изменения действий процесса при получении соответствующего сигнала. Параметр signum задает номер сигнала и может быть равен любому номеру, кроме SIGKILL и SIGSTOP. Если параметр act не равен нулю, то новое действие, связянное с сигналом signum, устанавливается соответственно act. Если oldact не равен нулю, то предыдущее действие записывается в oldact. Структура sigaction имеет следующий формат:

struct sigaction {

void (\*sa\_handler)(int);

void (\*sa\_sigaction)(int, siginfo\_t \*, void \*);

sigset\_t sa\_mask;

int sa\_flags;

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

}

Листинг 3. Приложение, реализующее обработку сигналов HUP, INT, ABRT

#include <signal.h>

#include <errno.h>

#include <stdio.h>

#include <string.h>

volatile sig\_atomic\_t signal\_number;

void handler(int signo) {

// signo автоматически блокируется функцией sigaction()

signal\_number = signo;

}

int main(int argc, char \*\*argv) {

sigset\_t set;

struct sigaction act;

sigemptyset(&set); // Создать пустой набор

sigaddset(&set, SIGHUP); // Добавить в набор SIGHUP

sigaddset(&set, SIGINT); // Добавить в набор SIGINT

sigaddset(&set, SIGABRT); // Добавить в набор SIGABRT

sigprocmask(SIG\_BLOCK, &set, NULL); // Заблокировать SIGINT

act.sa\_mask = set; // Настроить обработчик

act.sa\_handler = handler;

act.sa\_flags = 0;

sigaction(SIGHUP, &act, NULL); // Установить обработчик

sigaction(SIGINT, &act, NULL); // Установить обработчик

sigaction(SIGABRT, &act, NULL); // Установить обработчик

// Возможно, установить обработчики для других сигналов

// Восстановить пустой набор, допускает SIGINT

sigemptyset(&set);

while (1) {

sigsuspend(&set); // Ждать появления SIGINT

if (errno != EINTR) {

perror("error on sigsuspend");

return 1;

}

if (signal\_number == SIGHUP || signal\_number == SIGINT || signal\_number == SIGABRT) {

printf("signal: %s\n", strsignal(signal\_number));

// Обработка сигнала. SIGINT здесь снова блокируется

break;

}

}

printf("hi");

/\* ...любой другой код... \*/

return 0;

}

Результат выполнения листинга 3 приведен на рисунке 3.

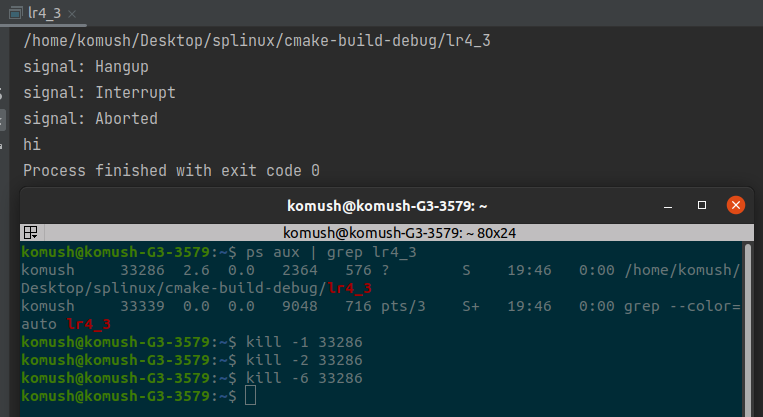


Рисунок 3. – Результат работы приложения

# Многопоточное приложение, состоящее из 4 потоков

В рамках разработки приложения был изучен порядок обработки сигналов в отдельных потоках. Обработка сигналов в каждом потоке производилась посредством системного вызова sigaction, рассмотренного ранее. Для потоков был создан класс, индивидуальные параметры каждого потока (обрабатываемый сигнал и имя) передаются в конструктор, создание экземпляров класса происходит отдельно в каждом потоке. Класс содержит методы Waiting и PrintPid. Первый отвечает за ожидание и обработку нужного сигнала, второй выводит pid потока на экран.

Листинг 4. Многопоточное приложение, обрабатывающее сигналы HUP, INT, ABRT

#include <signal.h>

#include <cerrno>

#include <cstdio>

#include <cstring>

#include <unistd.h>

#include <sys/wait.h>

#include <string>

#include <utility>

volatile sig\_atomic\_t signal\_number;

void handler(int signum) {

signal\_number = signum;

}

class Thread {

private:

sigset\_t set{};

int sig;

std::string name;

public:

Thread(int signum, std::string name) {

sig = signum;

this->name = std::move(name);

sigemptyset(&set);

sigaddset(&set, sig);

sigprocmask(SIG\_BLOCK, &set, nullptr);

struct sigaction act{};

act.sa\_mask = set;

act.sa\_handler = handler;

act.sa\_flags = 0;

sigaction(sig, &act, nullptr);

}

void Waiting() {

sigemptyset(&set);

while (true) {

sigsuspend(&set);

if (errno != EINTR) {

perror("error on suspend");

return;

}

if (signal\_number == sig)

printf("signal of %s: %s\n", name.c\_str(), strsignal(signal\_number));

}

}

void PrintPid() {

printf("%d is %s PID\n", getpid(), name.c\_str());

}

};

int main() {

int status;

pid\_t pid\_first = fork();

if (pid\_first == 0) {

Thread first(SIGHUP, "firsts");

first.PrintPid();

first.Waiting();

} else {

pid\_t pid\_second = fork();

if (pid\_second == 0) {

Thread second(SIGINT, "second");

second.PrintPid();

second.Waiting();

} else {

pid\_t pid\_third = fork();

if (pid\_third == 0) {

Thread third(SIGABRT, "third");

third.PrintPid();

third.Waiting();

}

}

}

wait(&status);

}

Результат выполнения листинга 4 приведен на рисунке 4.

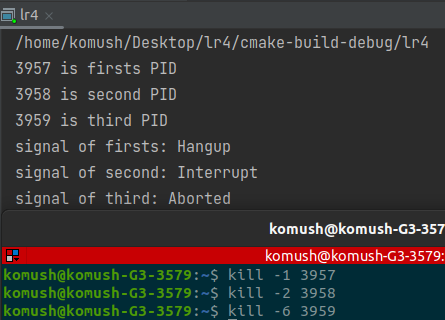


Рисунок 4. – Результат работы приложения

# Вывод

Разработано приложение, реализующее обработку сигналов HUP, INT, ABRT с помощью функции signal. Приложение информирует о получении соответствующего сигнала вне обработчика. Работа скрипта продемонстрирована посредством вызова команды kill.

Так же разработано приложение, реализующее обработку сигналов HUP, INT, ABRT с помощью сигнальной маски. Работа скрипта продемонстрирована посредством вызова команды kill.

Реализована безопасная обработка сигналов HUP, INT, ABRT с помощью системного вызова sigaction. Работа скрипта продемонстрирована посредством вызова команды kill.

Разработано приложение, состоящее из 4 потоков, где каждый поток обрабатывает только один сигнал: первый поток - HUP, второй - INT, третий - ABRT, главный поток - порождает дочерние потоки. Для каждого потока реализован отдельный класс.