**Синев Денис Евгеньевич**

**Пример 1**



Обработчик — функция, которая принимает номер полученного, сигнала, выполняет какие-то действия по его обработке и ничего не возвращает. В этом примере в функции main() устанавливается обработчик сигнала SIGINT с помощью функции signal(), который печатает сообщение о том, что он был вызван, и завершает выполнение процесса. Основная же программа запускает бесконечный цикл, а следовательно завершиться «сама» она не сможет.

Когда мы запустим эту программу, то она бесконечно будет ожидать команды. Прервать её выполнение можно только с помощью внешнего воздействия, т.е. с помощью отправки некоторого сигнала выполняющемуся процессу.



Нажмём Ctrl+C, отправив сигнал SIGINT

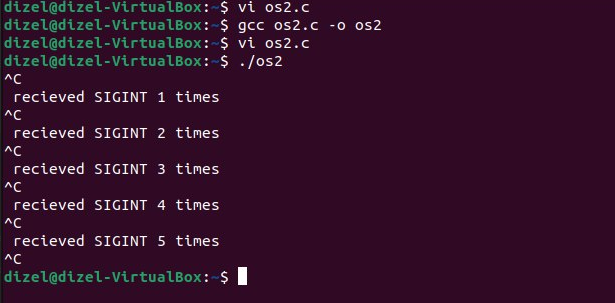


Сработал обработчик сигнала и процесс завершился.

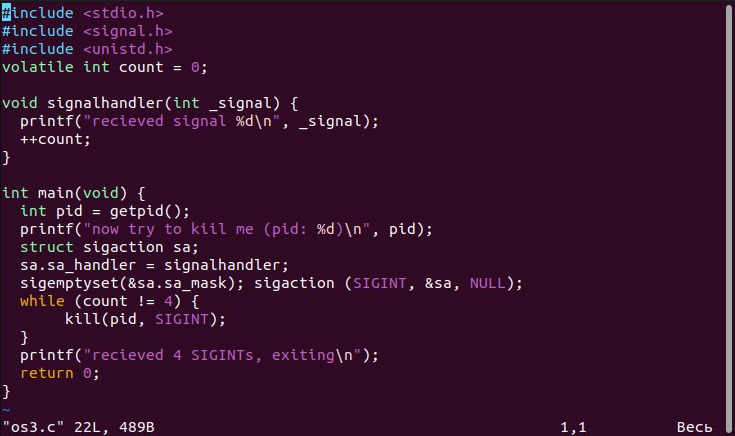
**Пример 2**

****

В этом примере обработчик сигнала будет выставлять следующий обработчик того же сигнала в зависимости от глобальной переменной count — пока её значение меньше 5 он будет оставаться таким же. Когда значение count станет равным 5, установится стандартный обработчик сигнала (SIG\_DFL). При этом каждый вызов signalhandler() увеличит count на 1. Программа завершилась только с 6-го раза, так как только после 5-й отправки сигнала, установился обработчик, который завершил процесс.



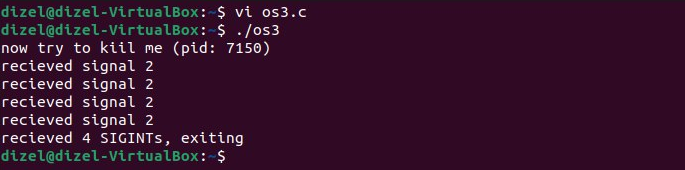
**Примеры 3 и 4**

****

Переменная count здесь объявлена как volatile — это означает, что любые оптимизации со стороны компилятора, связанные с этой переменной - отключаются. Это гарантирует, что разные части программы будут читать и изменять одну и ту же область памяти, и, таким образом, позволяет избежать ошибок.

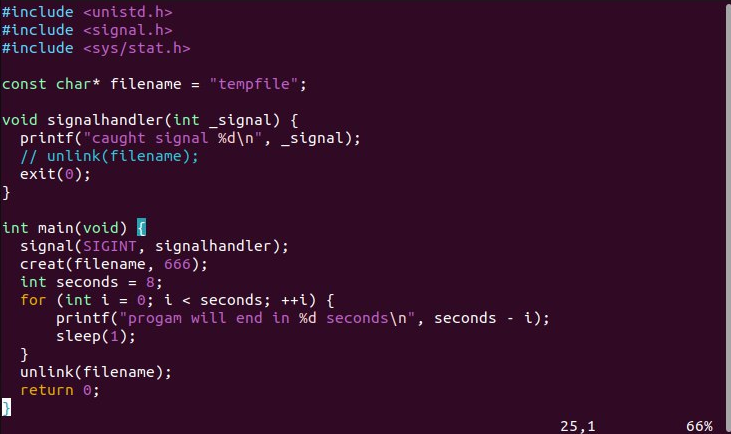
Установка обработчика сигнала теперь осуществляется, с помощью структуры и функции sigaction. Её использование гарантирует, что программа будет одинаково работать на всех системах. Для её использования необходимо объявить структуру sigaction, установить обработчик (sa.sa\_handler = signalhandler), инициализировать маску, состоящую из битов, обозначающих сигналы, которые заблокированы для обработки процессом (sigemptyset(&sa.sa\_mask)) и в конце установить эту структуру как обработчик сигнала SIGINT (sigaction(SIGINT, &sa, NULL).

Процесс сам отправляет себе 4 сигнала SIGINT с помощью вызова kill(pid, SIGINT), где pid — идентификатор выполняющего процесса.



Если тело цикла while мы оставим пустым, то нам самим (вручную) придется отправлять четыре сигнала.

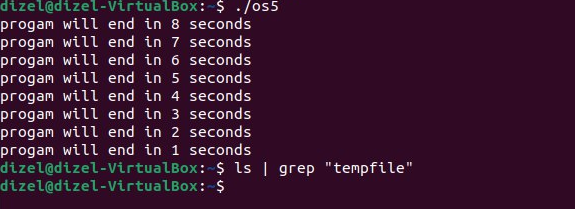
**Пример 5**

****

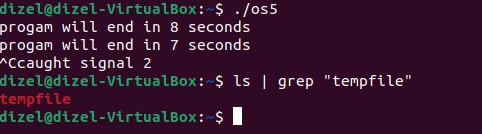
Данный пример показывает нам, что стоит писать код, освобождающий ресурсы внутри обработчика сигналов.

Создадим временный файл tempfile (с помощью фукнции creat), который предполагается удалить при завершении программы (unlink()). Обратим внимание, что пока обработчик сигнала SIGINT только печатает сообщение и завершает работу процесса.

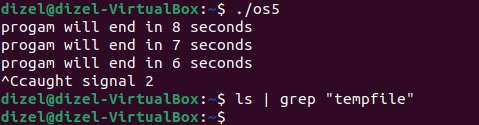
Запуская программу в первый раз мы не обнаружим временный файл



Но остановив процесс, мы обнаружим временный файл



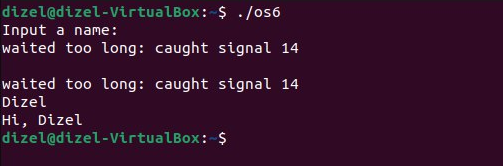
Если мы добавим вызов unlink() в обработчике сигнала, то временный файл уже удалится

****

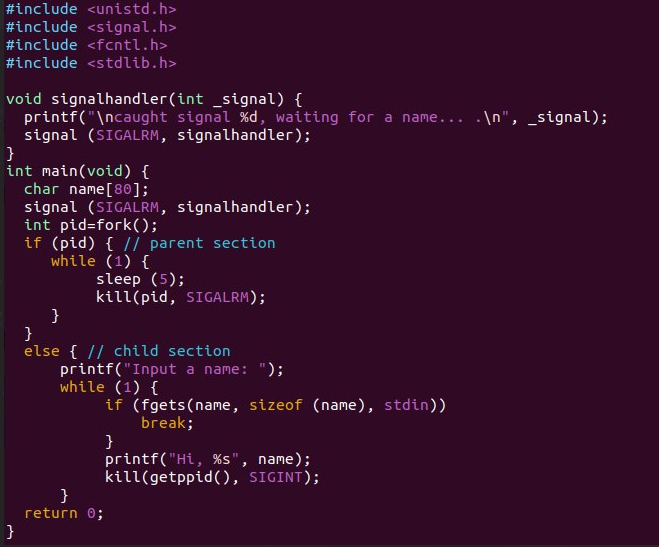
**Пример 6**

****

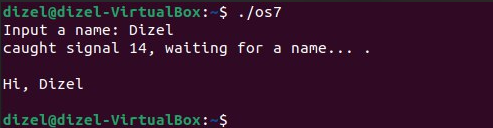
В данный примере показано, как можно посылать и обрабатывать сигнал SIGALRM — сигнал таймера. Функция alarm() выставляет таймер (в данном случае на 5 секунд), по истечению этого времени вызвавшему эту функцию процессу отправится сигнал SIGARM, для которого в нашей программе выставлен обработчик, который выводит сообщение и снова выставляет таймер на 5 секунд. Основная же программа ждёт ввода пользователя, напоминая каждые 5 секунд, что она всё еще ждёт.

****

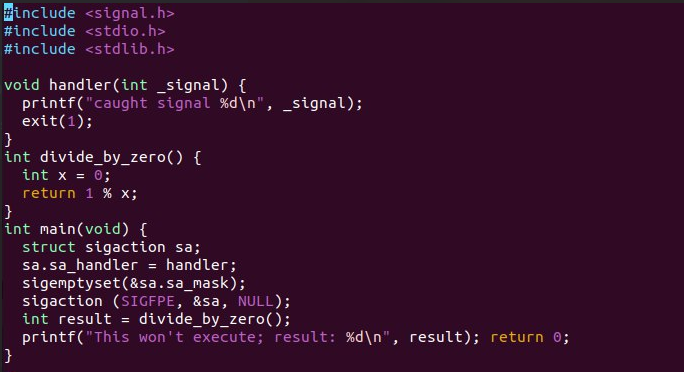
**Пример 7**

****

Тут выполняются те же действия, что и в примере 6. Теперь задачи разделены на 2 процесса. Родительский процесс каждые 5 секунд отправляет дочернему сигнал SIGALRM, а дочерний процесс, в свою очередь, ждёт ввода с консоли и реагирует на полученный сигнал таймера сообщением. После успешного ввода дочерний процесс завершает работу родителя (а вместе с этим и свою).



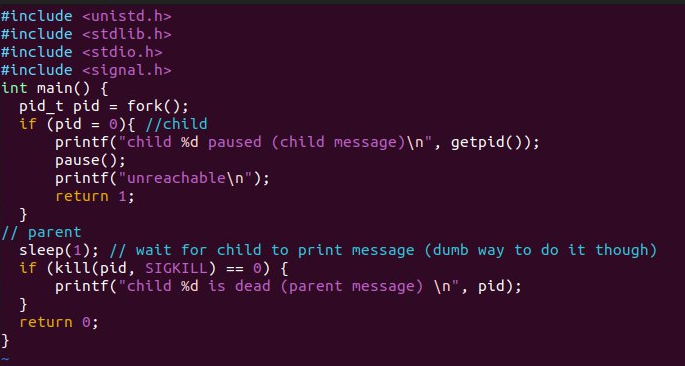
**Пример 8**

****

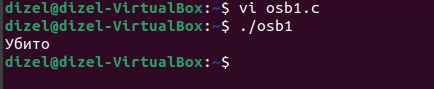
Данная программа выполняет деление на 0. В этом случае процессу будет послан сигнал SIGFPE.



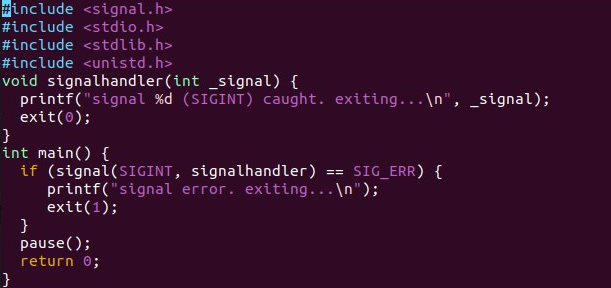
**Пример из книги 1**

****

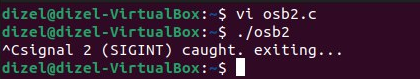
Этот пример ещё раз демонстрирует возможность отправки сигнала от родительского процесса к дочернему. Создается дочерний процесс — он печатает сообщение и ставится на паузу (pause()) до получения некоторого сигнала. Родительский же процесс отправляет дочернему сигнал SIGINT, завершая его работу.



**Пример из книги 2**

****

Здесь основной процесс ставиться на паузу и ждёт сигнал. Так же в этом примере для SIGINT выставлен свой обработчик. Если же ошибка произошла, то функция вернёт SIG\_ERR.

****

**Пример из книги 3**

****

hadler1 сохраняет значение errno, а затем в конце восстанавливает его. Это нужно, так как код внутри обработчика может изменять эту глобальную переменную, изменяя номер последнего полученного исключения. Это не очень правильно, так как обработчики сигналов не являются частью основной программы, а значит ей не нужно знать об ошибках, произошедших во время прерывания на обработку сигнала.

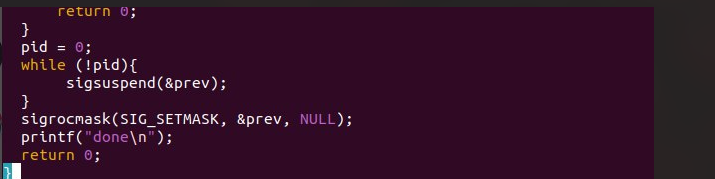
Также здесь наглядно демонстрируется тот факт, что сигналы одного типа не ставятся в очередь обработки (этой очереди просто нет: если какой-то сигнал пришёл вторым, до того как первый начал обрабатываться, то он просто теряется).

В начале handler1 ставится как обработчик сигнала SIGCHLD: этот сигнал посылается родительскому процессу и сообщает о том, что какой-то его потомок завершил работу. Далее создаются три дочерних процесса, которые печатают одно сообщение и тут же завершают работу. Родитель же ждёт ввод с консоли и входит в бесконечный цикл.

Обработчик handler1 убеждается, что один из процессов-потомков действительно завершил работу (вызов waitpid с аргументом -1), выводит об этом сообщение, а затем намеренно ждёт одну секунду, чтобы создать вышеописанную ошибку с двумя сигналами. В итоге получится что, два из трёх сигналов о завершении работы потомков пройдут обработку, а третий проигнорируется.

**Пример из книги 4**

****

****

Здесь функция handler\_t замена встроенной функции signal(). Она принимает те же аргументы, а на выходе мы имеем тот же результат, но внутри эта процедура использует sigaction. Таким образом, мы можем как бы использовать встроенный signal(), но по факту будем использовать sigaction.

Затем функция sigchild\_handler является обработичком сигналов. Объявлена глобальная переменная pid типа sig\_atomic\_t — это целое число с операциями, некоторые из которых атомарные. Такой тип необходим, для корректной асинхронной работы нескольких процессов, делящих эту переменную.

Обработчик для SIGCHLD изменяет глобальную переменную pid, записывая в неё идентификатор завершённого дочернего процесса. Обработчик для SIGINT ничего не делает.

В основной функции идея в том, чтобы показать, как явно блокировать некоторые сигналы и как это можно использовать. В начале инициализируется mask, в которой поднимается бит, отвечающий за сигнал SIGCHLD. Затем эта маска используется, чтобы заблокировать этот сигнал. Для этого используется вызов sigprocmask(SIG\_BLOCK, &mask, &prev), SIG\_BLOCK означает блокировку сигналов из mask (в нашем случае только SIGCHLD), при это предыдущий набор заблокированный сигналов сохранится в prev.

Затем создаётся дочерний процесс и сразу же завершается родительский процесс ждёт пока обработчик sigchld\_handler изменит pid. После чего Восстанавливает изначальную маску битов сигналов (SIG\_SETMASK), по сути отменяя блокировку SIGCHLD.

Вызов sigsuspend(&prev) по сути является атомарным аналогом последовательности команд sigprocmask(SIG\_SETMASK, &prev, &another\_prev); pause();

sigprocmask(SIG\_SETMASK, &another\_prev, NULL);

Иначе говоря, временно ставится другая маска сигналов, затем процесс останавливается до получения сигнала, после чего возвращается старая маска. Поместив этот вызов в цикл, Получаем самый правильный и безопасный способ ожидания сигнала от дочернего процесса. Мы избегаем разного рода ошибок, связанных с «очередью»