Міністерство освіти і науки України

Харківський національний університет імені В.Н. Каразіна

Факультет комп’ютерних наук

# ЛАБОРАТОРНА РОБОТА № 9

з дисципліни «Операційні системи»

Тема: «Процеси. Сигнали»

Виконав:

студент 3 курсу

групи КС-32

Безрук Юрій Русланович

Перевірив: Споров О.Є.

Харків – 2020

Целью данной работы является ознакомление с сигналами – традиционным механизмом общения ядра и процессов пользователя, который может быть использован для организации простейшего межпроцессного взаимодействия.

# ХОД РАБОТЫ

**Задание 1**

В задании один нужно поэкспериментировать с назначением пользовательских реакций на различные сигналы. Реализовано две версии программы, код которых размещен в файлах signal.c и sigaction.c.

Первая версия:

Для задания возможности реагирования на сигналы создаем функцию-обработчик согласно сигнатуре типа sighandler\_t, в которой проверяем значение параметра signo, который представляет собой номер сигнала, для обработки которого процессом была вызвана функция-обработчик. Согласно условию задания, в случае получения сигналов SIGINT или SIGTERM, просто информируем об этом, сопоставляя номер сигнала с массивом строк-названий сигналов char\*\* sys\_siglist. В случае обработки сигнала SIGUSR1 завершаем работу программы вызовом функции exit(). Так же добавляем альтернативный исход на всякий случай – информацию о том, что обработчик поймал какой то другой сигнал, на который мы не рассчитывали.

В функции main() регистрируем объявленную функцию-обработчик как обработчик для сигналов SIGINT, SIGTERM и SIGUSR1 при помощи функции signal(int signo, sighandler\_t handler). Кроме того, согласно условию задания, устанавливаем поведение по умолчанию для сигнала SIGPROF, устанавливая вместо обработчика специальный флаг SIG\_DFL, а для сигнала SIGHUP устанавливаем игнорирование, на этот раз при помощи флага SIG\_IGN. После этого в бесконечном цикле вызываем функцию pause(), которая блокирует выполнение программы до получения какого-либо сигнала. Таким образом, программа завершится только в том случае, если ей будет отправлен сигнал на завершение (как пользовательский установленный нами SIGUSR1, так и специальный SIGSTOP/SIGKILL/другие сигналы, поведение по умолчанию для которых определено как завершение).

Вторая версия:

Во второй версии в отличие от прошлой, вместо функции signal(int signo, sighandler\_t handler) для регистрации обработчиков используется функция sigaction(int signo, struct sigaction\* act, struct sigaction\* oldact), которая позволяет произвести более тонкую настройку обработчика сигналов. В структуре, на которую указывает act, можно указать два различных обработчика: sa\_handler(int), который будет работать так же, как и предыдущий, и расширенный – sa\_sigaction(int, siginfo\_t\*, void\*), который будет получать дополнительную информацию о сигнале, вызвавшем его, а не только его номер. Также при помощи этой структуры можно указать маску сигналов, которые нужно блокировать на время выполнения обработчика сигналов и служебные флаги.

Т.к. обычный обработчик применялся в прошлом варианте программы, в этот раз используем расширенный обработчик. Принцип его работы остается таким же, за исключением того, что мы теперь можем использовать служебную информацию о сигнале. Поэтому, для сигналов SIGINT и SIGTERM выведем так же номер процесса, который его отправил.

В функции main() всё так же регистрируем обработчики, но порядок теперь становится другой. Для того, чтобы показать на что еще способна функция sigaction, создадим сигнальную маску – набор сигналов, который будет блокироваться на время выполнения обработчика. Объявляем набор сигналов типа sigset\_t. Далее при помощи функции sigemptyset() инициализируем этот набор как пустой, после чего добавляем к нему все сигналы, поведение которых мы определяем, кроме SIGHUP, т.к. мы и так планируем установить постоянное игнорирование этого сигнала. Добавление к набору производится при помощи функции sigaddset(). Т.е. идея такая: игнорировать любой другой сигнал из тех что мы определили во время обработки одного из них.

Далее создаем переменную структуры struct sigaction act, в поле sa\_mask которой устанавливаем значение нашего набора сигналов. Расширенный обработчик устанавливаем в поле sa\_sigaction. Для того, чтобы был использован именно он, а не обычный, в поле sa\_flags устанавливаем значение SA\_SIGINFO.

После этого регистрируем обработчик для сигналов SIGINT, SIGTERM, SIGUSR1 при помощи функции sigaction(), где в качестве первого аргумента передаем код сигнала, в качестве второго – адрес настроенной структуры, а в качестве третьего – NULL, т.к. старое значение этих параметров нас не интересует в данной задаче.

Для установки поведения по умолчанию для сигнала SIGPROF, зануляем флаг в структуре, чтобы обработчиком выступало поле простого хендлера, которому и присваиваем значение SIG\_DFL. Для игнорирования таким же образом.

Для проверки работы программы запускаем ее в терминале, а из другого терминала отправляем сигналы консольной командой kill.

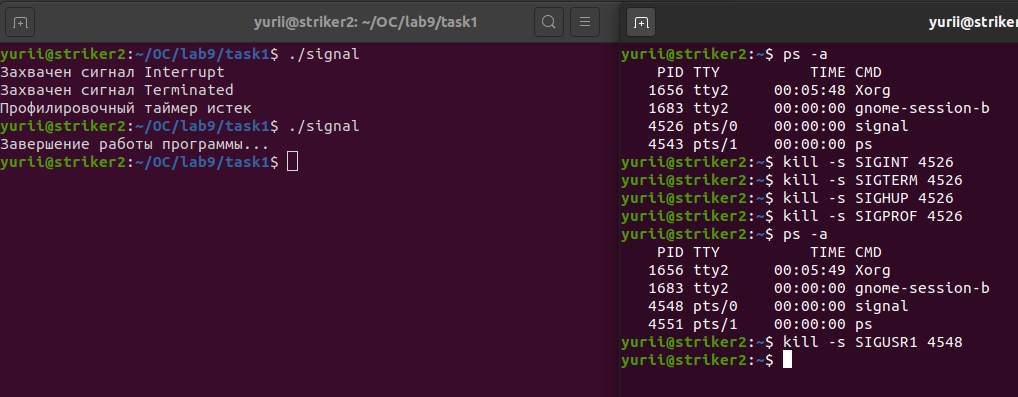


Рисунок 1 – пример работы обработчика сигналов зарегистрированных при помощи signal()

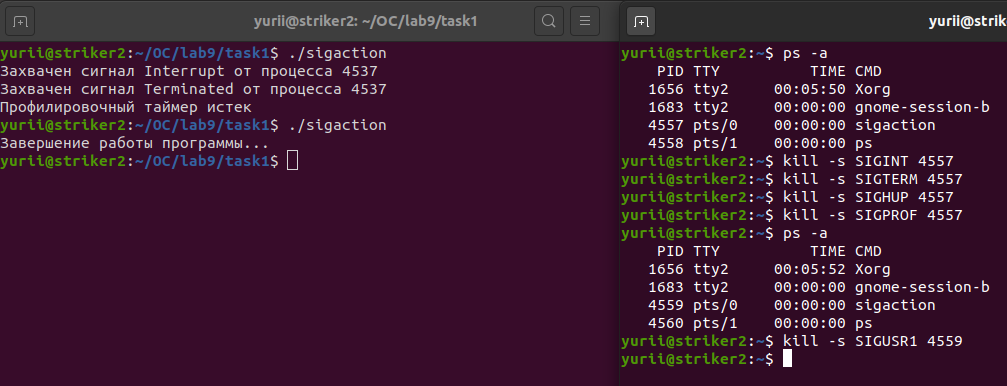


Рисунок 2 – пример работы обработчика сигналов зарегистрированных при помощи sigaction()

**Задание 2**

В задании 2 требуется написать программу-будильник, которая будет получать требуемый интервал времени в секундах и текстовое сообщение при запуске из консоли, завершает основной процесс, а в дочернем выводит сообщение через заданное количество секунд.

Для этого создаём обычный обработчик, который будет обрабатывать сигнал SIGALRM и выводить сообщение в случае его получения. Переменную сообщения объявляем глобально. Так же дополнительно регистрируем обработчик завершения, что бы память, выделенная под текстового сообщение, автоматически очищалась при любом завершении программы. Задержку по умолчанию для будильника устанавливаем равной 10 секундам, а сообщение – «привет, мир». Функцией signal() регистрируем обработчик сигнала, после чего создаем дочерний процесс при помощи функции fork() – процесс-родитель завершаем, а в дочернем процессе вызываем функцию alarm(), которая через переданное ей количество времени сгенерирует сигнал SIGALRM. Т.к. вызов этой функции не является блокирующим, заходим в бесконечный цикл с вызовом функции pause() (цикл нужен, что бы никакой другой сигнал кроме завершающих не смог продолжить выполнение программы).

Дочерний процесс в этой программе нужен для того, что бы работа в терминале не приостанавливалась на время выполнения программы. Таким образом, данная программа всегда выполняется в фоне, т.к. основной процесс, который создается при ее запуске, почти сразу же завершается, а ожидание будильника происходит на заднем плане потомком.

Пример работы программы:

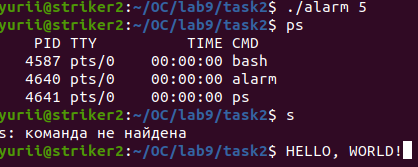


Рисунок 3 – будильник сработал и вставил текст сообщения в строку терминала через 5 секунд

**Задание 3**

В задании 3 требуется реализовать простейший вариант синхронизации процессов при помощи сигналов. Процессы должны поочередно выводить сообщения в консоль, посылая друг другу сигнал SIGUSR1.

Реализуем это таким образом:

Обработчик сигналов настраиваем на прием сигнала SIGUSR1. В случае его получения он выводит своё сообщение, после чего функцией kill() отправляет такой же сигнал другому процессу, идентификатор которого данный процесс уже знает (родительский процесс получает его при создании дочернего, а дочерний – вызовом функции getppid()).

В функции main() регистрируем обработчик, и вызываем fork(). В случае дочернего процесса производим дополнительные действия: получаем идентификатор родительского процесса, изменяем сообщение и, для того что бы начать процесс обмена, посылаем сигнал родительскому процессу. Как альтернативу, можно послать сигнал и самому себе, используя функцию raise(). Ну и после этого оба процесса заходят в бесконечный цикл ожидания прихода сигнала, который они будут обрабатывать. Для подстраховки, по аналогии с мьютексами, было решено блокировать сигналы SIGUSR1 во время вывода сообщения процессом при помощи сигнальной маски. набор предварительно инициализируется одним сигналом, после чего, в обработчике, перед началом вывода сообщения устанавливается блокировка этого сигнала при помощи функции sigprocmask(int how,const sigset\_t\* set, sigset\_t\* oldset). Набор сигналов передаётся во второй аргумент, а в первый указывается флаг SIG\_BLOCK, что означает, что сигналы добавляются к тем, что уже есть в маске в случае наличия таковых. В третьем аргументе передается NULL за ненадобностью использования старой маски. После вывода сообщения блокировка снимается аналогичным вызовом, но уже с флагом SIG\_UNBLOCK, который удаляет сигналы набора из существующей маски процесса.



Рисунок 4 – фрагмент вывода программы

**Задание 4**

Задание 4 рассчитано на использование возможности передачи служебных данных при помощи сигналов. В программе принимают участие два процесса. Родительский заданное количество раз посылает дочернему сигнал SIGUSR1 с дополнительной информацией – номером вызова. Дочерний процесс обрабатывает сигнал и выводит полученный номер, текстовое представление сигнала и дополнительную информацию. После отправки всех сообщений основной процесс отправляет дочернему сигнал SIGTERM, ожидает его завершения и завершается сам.

Для реализации данной задачи используется расширенный обработчик сигналов, который извлекает служебную информацию из структуры siginfo, как переданный номер вызова, так и идентификатор вызывающего процесса. Номер вызова передается через поле si->si\_value, которое является полем типа обьединение: union sigval, что означает, что только одно из полей структуры может одновременно быть проинициализировано. А полей там всего два: либо целочисленное значение, либо неопределенный указатель.

Таким образом, дочерний поток после создания входит в цикл ожидания сигналов, а родительский – инициализирует объединение типа sigval, и циклически производит отправку сигналов, передавая в качестве данных номер вызова. Выполняется это при помощи функции sigqueue(). Для того, чтобы процесс-потомок успевал их обрабатывать, добавлена небольшая задержка в 0,1 секунды между вызовами. После завершения отправки всех сигналов, родительский процесс отправляет дочернему сигнал SIGTERM, принуждая того завершиться, ожидает его завершения при помощи функции waitpid() и завершается сам.

Количество сообщений передается при запуске с консоли, по умолчанию – 5.

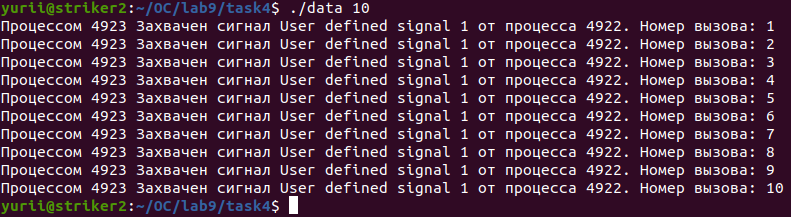


Рисунок 5 - пример передачи данных процессом через сигнал

# ВЫВОДЫ

Таким образом, в ходе выполнения данной работы были рассмотрены основные возможности управления сигналами в Linux, такие, как обработчики сигналов, сигнальные маски, очереди и наборы сигналов, отправка сигналов и служебные данные сигнала. Исходные, исполняемые и файлы сборки помещен в архив.