# 1. ОСНОВНЫЕ КОМАНДЫ ОС UNIX

Операционная система ОС ***Linux*** создана на основе ОС ***UNIX*** и во многом имеет схожую структуру и систему команд. Пользователь может работать в текстовом режиме с помощью командной строки, или с использованием графического интерфейса ***X Window*** и одного из менеджеров рабочего стола (например, ***KDE*** или ***GNOME***). Причем, одновременно в системе могут работать 7 пользователей (6- в текстовом режиме консоли и 1 – в графическом режиме), переключение между пользователями осуществляется по нажатию клавиш:

.

***Ctrl***

***Alt***

***F1***

***Ctrl***

***Alt***

***F7***

В табл. 1 приведены основные команды системы

Таблица 1

|  |  |  |  |
| --- | --- | --- | --- |
| Команда | Аргументы/ключи | Пример | Описание |
| ***dir*** | каталог | ***dir***  ***dir /home*** | Выводит на консоль содержимое каталога |
| ***ls*** | ***-all***  и другие (см. man) | ***ls -all*** | Выводит на консоль содержимое каталога |
| ***ps*** | ***-a***  ***-x*** и другие (см. man) | ***ps -a*** | Выводит на консоль список процессов |
| ***mkdir*** | имя каталога | ***mkdir stud11*** | Создает каталог |
| ***rmdir*** | имя каталога | ***rmdir stud11*** | Удаляет каталог |
| ***rm*** | файл | ***rm myfile1*** | Удаляет файл |
| ***mv*** | файл новое имя | ***mv myfile1 myf1*** | Переименование файла |
| ***cat*** | файл | ***cat 1.txt*** | Вывод файла на консоль |
| ***cd*** | имя каталога | ***cd home*** | Переход по каталогам |
| ***grep*** | (см. man) | ***grep "^a" "words.txt"*** | Поиск строки в файле |
| ***kill*** | ***pid*** процесса | ***kill 12045*** | Уничтожает процесс |
| ***top*** |  |  | Выводит на консоль список процессов |
| ***htop*** |  |  | Выводит на консоль полный список запущенных процессов |
| ***su*** |  |  | Переход в режим root |
| ***chmod*** | права доступа файл | ***chmod 777 1.txt*** | Изменение прав доступа к файлам |
| ***mount*** | устройство каталог | ***mount /dev/cdrom /MyCD*** | Монтирование устройств |
| ***dd*** | ***if=***файл ***of=***файл ***bs=n count=n*** | ***dd if=/dev/sda1***  ***of=/F.bin bs=512 count=1*** | Копирование побайтное |
| ***ln*** | файл1 файл2  ***-l*** | ***ln файл1 файл2***  ***ln –l файл1 файл2*** | Создать жёсткую или символическую ссылку на файл |
| ***uname*** | ***-a*** | ***uname –a*** | Информация о системе |
| ***find*** | dir файл | ***find /home –name a1.txt*** | Поиск файлов |
| ***man*** |  | ***man fgetc*** | Справка по системе |
|  |  |  |  |
| ***info*** |  | ***info fgetc*** | Справка по системе |
| ***apt-get*** | apt-get install пакет |  | Инсталляция программного пакета |
| ***apt-get*** |  | **apt-get update**  **apt-get upgrade**  ***apt-get install*** | Обновление сведений о репозиториях программных продуктов и установка обновлений |
| ***cp*** |  |  |  |

Linux и Windows используют различные файловые системы для хранения и организации доступа к информации на дисках. В Linux используются файловые системы- ***Ext2/Ext3/Ext4***, ***RaiserFS, FFS и*** другие. Все файловые системы имеют поддержку ***журналирования***. ***Журналируемая*** файловая система сначала записывает изменения, которые она будет проводить в отдельную часть файловой системы (***журнал***) и только потом вносит необходимые изменения в остальную часть файловой системы. После удачного выполнения всех транзакций, записи удаляются из ***журнала***. Это обеспечивает лучшее сохранение целостности системы и уменьшает вероятность потери данных. Следует отметить, что ***Linux*** поддерживает доступ к ***Windows***-разделам.

Файловая система ***Linux*** имеет лишь один корневой каталог, который обозначается косой чертой ( ***/*** ). В файловой структуре ***Linux*** нет дисков ***A, B, C, D*** …., а есть только каталоги. В ***Linux*** различаются прописные и строчные буквы в командах, именах файлов и каталогов. В ***Windows*** у каждого файла существует лишь одно имя, в ***Linux*** их может быть много. Это – «***жесткие***» ссылки, которые указывают непосредственно на индексный дескриптор файла. Жесткая ссылка – это один из принципов организации файловой системы ***Linux***.

Структура каталогов ОС ***Linux*** представлена в табл. 1. Есть также несколько полезных сокращений для имен каталогов:

* Одиночная точка (**.**) обозначает текущий рабочий каталог.
* Две точки (**..**) обозначают родительский каталог текущего рабочего.
* Тильда (**~**) обозначает домашний каталог пользователя (обычно это каталог, который является текущим рабочим при запуске Bash).

Таблица 1

|  |  |
| --- | --- |
| ***/*** | Корневой каталог |
| ***/bin*** | Содержит исполняемые файлы самых необходимых для работы системы программ. Каталог ***/bin*** не содержит подкаталогов. |
| ***/boot*** | Здесь находятся само ядро системы (файл ***vmlinuz***-...) и файлы, необходимые для его загрузки. |
| ***/dev*** | Каталог ***/dev*** содержит файлы устройств (драйверы). |
| ***/etc*** | Это каталог конфигурационных файлов, т. е. файлов, содержащих информацию о настройках системы (например, настройки программ). |
| ***/home*** | Содержит домашние каталоги пользователей системы. |
| ***/lib*** | Здесь находятся библиотеки (функции, необходимые многим программам). |
| ***/media*** | Содержит подкаталоги, которые используются как точки монтирования для сменных устройств. |
| ***/mnt*** | Данный каталог (или его подкаталоги) может служить точкой монтирования для временно подключаемых файловых систем. |
| ***/proc*** | Содержит файлы с информацией о выполняющихся в системе процессах. |
| ***/root*** | Это домашний каталог администратора системы. |
| ***/sbin*** | Содержит исполняемые программы, как и каталог ***/bin***. Однако использовать программы, находящиеся в этом каталоге может только администратор системы (***root***). |
| ***/tmp*** | Каталог для временных файлов, хранящих промежуточные данные, необходимых для работы тех или иных программ, и удаляющиеся после завершения работы программ. |
| ***/usr*** | Каталог для большинства программ, которые не имеют значения для загрузки системы. Структура этого каталога фактически дублирует структуру корневого каталога. |
| ***/var*** | Содержит данные, которые были получены в процессе работы одних программ и должны быть переданы другим, и файлы журналов со сведениями о работе системы. |

# 2. ЛАБОРАТОРНЫЕ РАБОТЫ

# ЛАБОРАТОРНАЯ РАБОТА №1 УПРАВЛЕНИЕ ОС LINUX, ИНТЕРПРЕТАТОР BASH

Цель работы – изучить основные объекты, команды, типы данных и операторы управления интерпретатора BASH; создать скрипт-файл.

**Теоретическая часть**

***Bash*** - это ***sh***-совместимый интерпретатор командного языка, выполняющий команды, прочитанные со стандартного входного потока или из файла. ***Скрипт-файл*** – это обычный текстовый файл, содержащий последовательность команд ***bash***, для которого установлены права на выполнение. Пример скрипта, выводящего содержимое текущего каталога на консоль и в файл:

***#!/bin/bash***

***dir***

***dir > 1.txt***

Следующие переменные используются командным интерпретатором.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ***$0, $1, $2, $3…*** | | | Значения аргументов командной строки при запуске скрипта. Где ***$0***-имя самого файла скрипта, ***$1***- первый аргумент, ***$2***- второй аргумент, и т.д. | |  |
| ***$@*** | | | Все аргументы командной строки начиная с 1-го | |  |
| ***$?*** | Код возврата последней команды | | | | |
| ***$\**** | Все аргументы командной строки начиная с 1-го | | | | |
| ***$#***­­ | | Число переданных аргументов | |  |  |

Пример простого скрипта, выводящего на консоль и в файл содержимое каталога, где имя каталога передаётся скрипту в качестве аргументов при запуске:

Запуск скрипта: ***./home/stud 1.sh***

Скрипт:

***#!/bin/bash***

***dir $1***

***dir $1 > 1.txt***

Можно создать собственную переменную и присвоить ей значение:

***A=121***

***A=”121”***

***let A=121***

***let “A=А+1”***

Вывод значения на консоль: ***echo $A***

Организация циклов:

1. ***for var1 in list***

***do***

***…***

***done***

2. ***while exp***

***…***

***end***

3. ***until exp // аналог do-while***

***do***

***…***

***done***

Проверка условия: ***test[expr]***

где ***expr***: а) для строк: ***S1 = S2*** ***S1***содержит ***S2***

***S1 != S2******S1***не содержит ***S2***

***-n S1*** если длина ***S1 >0***

***-z S1*** если длина ***S1 =0***

б) целые ***i1***и ***i2***

***i1 – ge i2***

***i1 – gt i2***

***i1 – ie i2***

***i1 – et i2***

***i1 – nt i2***

в) файлы

***-d name\_file*** является ли файл каталогом

***-f name\_file*** является ли файл обычным файлом

***-r name\_file*** доступен ли файл для чтения

***-s name\_file*** имеет ли файл ненулевую длину

***-w name\_file*** доступен ли файл для записи

***-x name\_file*** является ли файл исполняемым

г) логически операции

***!exp*** логическое отрицание (не)

***exp1 –a exp2*** умножение условий (и)

***exp1 –o exp2*** сложение условий (или)

Проверка условия: ***if [expr ]***

***then com 1*** Если условие ***expr=true*** то команда

***…***

***(elif expr2***

***…***

***)***

***else***

***…***

***fi***

Проверка нескольких условий: ***case string1 in***

***str 1)***

***…***

***; ;***

***str 2)***

***…***

***; ;***

***str 3)***

***…***

***; ;***

***\*)***  // default

***…***

***; ;***

***esac***

Функция пользователя: ***fname2 (arg1,arg2...argN)***

***{  
commands  
}***

***if [ -d “/home” ]***

***then***

***echo "Yes"***

***else***

***echo "No"***

***fi***

**Порядок выполнения работы**

1. Изучить теоретическую часть лабораторной работы.
2. В консольном режиме создать, используя команды из табл.1, в ***домашней папке*** подкаталог: ***/номер\_группы/ФИО\_студента,*** где в дальнейшем будут храниться все файлы студента. Перейти в корневой каталог и вывести его содержимое используя команды ***dir*** и ***ls –all*** , проанализировать различия.
3. Проверить действие команд ***ps***, ***ps –x, top, htop***. Найти в справочной системе используя команду ***man*** справку по функциям ***fprintf***, ***fputc*** и команде ***ls***.
4. В текстовом редакторе ***joe*** (вызов: ***joe 1.c***) написать программу ***1.c***, выводящую на экран фразу ***“HELLO SUSE Linux”***. Компилировать полученную программу компилятором ***gcc***: ***gcc 1.c –o 1.exe***. Запустить полученный файл ***1.exe*** на выполнение: ./1.EXE
5. Написать скрипт, выводящий на консоль и в файл все аргументы командной строки.
6. Написать скрипт, выводящий в файл (имя файла задаётся пользователем в качестве первого аргумента командной строки) имена всех файлов с заданным расширением (третий аргумент командной строки) из заданного каталога (имя каталога задаётся пользователем в качестве второго аргумента командной строки).
7. Написать скрипт, компилирующий и запускающий программу (имя исходного файла и ***exe-*** файла результата задаётся пользователем в качестве аргументов командной строки). В случае ошибок при компиляции вывести на консоль сообщение об ошибках и не запускать программу на выполнение.

## Варианты индивидуальных заданий

1. Написать скрипт для поиска файлов заданного размера в заданном каталоге (имя каталога задаётся пользователем в качестве третьего аргумента командной строки). Диапазон (мин.- мах.) размеров файлов задаётся пользователем в качестве первого и второго аргумента командной строки. Вывести на консоль первые найденные 20 файлов в виде: полный путь, имя файла, его размер.
2. Написать скрипт с использованием цикла ***for***, выводящий на консоль размеры и права доступа для всех файлов в заданном каталоге и всех его подкаталогах (имя каталога задается пользователем в качестве первого аргумента командной строки). На консоль выводится общее число просмотренных файлов.
3. Написать скрипт для поиска заданной пользователем строки во всех файлах заданного каталога и всех его подкаталогов (строка и имя каталога задаются пользователем в качестве первого и второго аргумента командной строки). На консоль выводятся полный путь и имена файлов, в содержимом которых присутствует заданная строка, и их размер. Если к какому-либо каталогу нет доступа, необходимо вывести соответствующее сообщение и продолжить выполнение.
4. Написать скрипт поиска одинаковых по содержимому файлов в двух каталогов, например, ***Dir1*** и ***Dir2***. Пользователь задаёт имена ***Dir1*** и ***Dir2*** в качестве первого и второго аргумента командной строки. В результате работы программы файлы, имеющиеся в ***Dir1***, сравниваются с файлами в ***Dir2*** по их содержимому. На экран выводятся число просмотренных файлов и результаты сравнения.
5. Написать скрипт, находящий в заданном каталоге и всех его подкаталогах все файлы, владельцем которых является заданный пользователь. Имя владельца и каталог задаются пользователем в качестве первого и второго аргумента командной строки. Скрипт выводит результаты в файл (третий аргумент командной строки) в виде полный путь, имя файла, его размер. На консоль выводится общее число просмотренных файлов.
6. Написать скрипт, находящий в заданном каталоге и всех его подкаталогах все файлы заданного размера. Диапазон (мин.- мах.) размеров файлов задаётся пользователем в качестве первого и второго аргумента командной строки. Скрипт выводит результаты поиска в файл (третий аргумент командной строки) в виде: полный путь, имя файла, его размер. На консоль выводится общее число просмотренных файлов.
7. Написать скрипт подсчитывающий суммарный размер файлов в заданном каталоге и отдельно во всех его подкаталогах (имя каталога задаётся пользователем в качестве аргумента командной строки). Скрипт выводит результаты подсчёта в файл (второй аргумент командной строки) в виде подкаталог (полный путь), суммарный размер файлов, число файлов в данном подкаталоге.
8. Написать скрипт, находящий в заданном каталоге и всех его подкаталогах все файлы заданного расширения и создающий для каждого найденного файла жесткую ссылку в заданном каталоге. Расширение файла и каталог для жестких ссылок задаются в качестве первого и второго аргумента командной строки.
9. Написать скрипт, находящий все каталоги и подкаталоги начиная с заданного каталога и ниже на заданной глубине вложенности (аргументы 1 и 2 командной строки). Скрипт выводит результаты в файл (третий аргумент командной строки) в виде полный путь, количество файлов в каталоге. На консоль выводится общее число просмотренных каталогов.
10. Написать скрипт, находящий все дубликаты (с одинаковым содержимым) файлов в заданном диапазоне размеров от ***N1*** до ***N2*** (***N1, N2*** задаются в аргументах командной строки), начиная с исходного каталога и ниже. Имя исходного каталога задаётся пользователем в качестве первого аргумента командной строки.
11. Написать скрипт, считающий для заданного каталога и всех его подкаталогов суммарный размер и количество файлов в заданном диапазоне размеров файлов (имя каталога задаётся пользователем в качестве аргумента командной строки, 2,3 аргументы командной строки диапазон размеров). Результаты выводятся на консоль в виде полный путь, количество файлов, суммарный размер.

# ЛАБОРАТОРНАЯ РАБОТА №2 *РАБОТА С ФАЙЛАМИ И КАТАЛОГАМИ ОС UNIX*

Цель работы – изучить основные системные вызовы и функции в ОС UNIX для работы с файлами и каталогами.

**Теоретическая часть**

Функция ***int main( int argc , char \*argv[ ] , char \*envp[ ] );***

Данное объявление позволяет удобно передавать аргументы командной строки и переменные окружения. Определение аргументов:

***argc*** - количество аргументов, которые содержатся в ***argv[]*** (всегда больше либо равен 1);

***argv*** - в массиве строки представляют собой параметры из командной строки, введенные пользователем программы. По соглашению, **argv [0] –** это команда, которой была запущена программа, **argv[1]** – первый параметр из командной строки и так далее до argv[argc] – элемент, всегда равный NULL;

***envp*** - это массив строк, которые представляют собой переменные окружения. Массив заканчивается значением ***NULL***.

Для выполнения операций записи и чтения данных в существующем файле его следует открыть при помощи вызова ***open ()****.*

***int open (const char \*pathname, int flags, [mode\_t mode]);***

***FILE \* fopen (const char \*pathname, int flags, [mode\_t mode]);***

Второй аргумент системного вызова ***open*** *-* ***flag****s -* имеет целочисленный тип и определяет метод доступа. Параметр ***flags***принимает одно из значений, заданных постоянными в заголовочном файле ***fcnt1.h***. В файле определены три постоянных:

***O\_RDONLY*** – открыть файл только для чтения,

***O\_WRONLY*** – открыть файл только для записи,

***O\_RDWR*** – открыть файл для чтения и записи,

или ***“r”***, ***“w”***, ***“rw”*** для ***fopen().***

Третий параметр ***mode*** устанавливает права доступа к файлуи является необязательным, он используется только вместе с флагом ***O\_CREAT***. Пример создания нового файла:

***# include <sys/types.h>***

***# include <sys/stat.h>***

***# include <fcntl.h>***

***int fd1;***

***FILE \*F1;***

***F1=fopen (“Myfile2.txt”, “w”, 0644);***

***fd1=open (“Myfile1.txt”, O\_CREAT, 0644);***

Системные вызовы ***sta****t* и ***fstat*** позволяют процессу определить значения свойств в существующем файле.

***#include <sys/types.h>***

***#include <sys/stat.h>***

***struct stat buf;***

***int stat (const char \*pathname, struct stat \*buf);***

***int fstat (int filedes, struct stat \*buf);***

Пример:

***struct stat st1***;

***stat(“1.exe”, &st1);***

Где ***pathname*** – полное имя файла, ***buf*** –структура типа ***stat***. Эта структура после успешного вызова будет содержать связанную с файлом информацию.

Поля структуры stat включает следующие элементы:

***struct stat {***

***dev\_t st\_dev; /\**** логическое устройство, где находится файл ***\*/***

***ino\_t st\_ino; /\**** номер индексного дескриптора ***\*/***

***mode\_t st\_mode; /\**** права доступа к файлу ***\*/***

***nlink\_t st\_nlink; /\**** количество жестких ссылок на файл ***\*/***

***uid\_t st\_uid; /\**** ID пользователя-владельца ***\*/***

***gid\_t st\_gid; /\**** ID группы-владельца ***\*/***

***dev\_t st\_rdev; /\**** тип устройства ***\*/***

***off\_t st\_size; /\**** общий размер в байтах ***\*/***

***unsigned long st\_blksize; /\**** размер блока ввода-вывода ***\*/***

***unsigned long st\_blocks; /\**** число блоков, занимаемых файлом ***\*/***

***time\_t st\_atime; /\**** время последнего доступа ***\*/***

***time\_t st\_mtime; /\**** время последней модификации ***\*/***

***time\_t st\_ctime; /\**** время последнего изменения ***\*/***

***};***

Права доступа в ***Linux***. Права доступа к файлам представлены в виде последовательности бит, где каждый бит означает разрешение на запись (***w***), чтение (***r***) или выполнение (***x***). Права доступа записываются для владельца-создателя файла (***owner***); группы, к которой принадлежит владелец–создатель файла (***group***); и всех остальных (***other***). Например, при выводе команды ***dir*** запись типа:

-***rwx r-x r-w***  ***1.exe***

означает, что владелец файла ***1.exe*** имеет права на чтение, запись и выполнение, группа имеет права только на чтение и выполнение, все остальные имеют права только на чтение. В восьмеричном виде получится значение ***0754***. В действительности манипулирует файлами не сам пользователь, а запущенный им процесс. Для просмотра прав доступа можно использовать функцию ***stat***.

Для записи прав доступа служит функция ***chmod***:

***#include <sys/types.h>***

***#include <sys/stat.h>***

***int chmod(const char \*pathname, mode\_t mode);***

Пример: ***chmod(“1.exe”, 0777);***

Каталоги в ОС ***Linux*** –это особые файлы. Для открытия или закрытия каталогов существуют вызовы:

***#include <dirent.h>***

***DIR \*opendir (const char \*dirname);***

***int closedir( DIR \*dirptr);***

Для работы с каталогами существуют системные вызовы:

***int mkdir (const char \*pathname, mode\_t mode)*** – создание нового каталога,

***int rmdir(const char \*pathname)*** – удаление каталога. Первый параметр – имя создаваемого каталога, второй – права доступа:

***retval=mkdir(“/home/s1/t12/alex”,0777);***

***retval=rmdir(“/home/s1/t12/alex”);***

Заметим, что вызов ***rmdir(“/home/s1/t12/alex”)*** будет успешен, только если удаляемый каталог пуст, т.е. содержит записи “точка” (.) и “двойная точка” (..). Для чтения записей каталога существует вызов:

***struct dirent \*readdir(DIR \*dirptr);***

Структура dirent такова: ***struct dirent {***

***long d\_ino;***

***off\_t d\_off;***

***unsigned short d\_reclen;***

***char d\_name [];***

***};***

Поле ***d\_ino*** - это число, которое уникально для каждого файла в файловой системе. Значением поля ***d\_off*** служит смещение данного элемента в реальном каталоге. Поле ***d\_name*** есть начало массива символов, задающего имя элемента каталога. Данное имя ограничено нулевым байтом и может содержать не более ***MAXNAMLEN*** символов. Тем самым описываемая структура имеет переменную длину, хранящуюся в поле ***d\_reclen***.

Пример вызова:

***DIR \*dp;***

***struct dirent \*d;***

***d=readdir(dp);***

При первом вызове функция ***readdir*** в структуру ***dirent*** будет считана первая запись каталога. После прочтения всего каталога в результате последующих вызовов ***readdir*** будет возвращено значение ***NULL***. Для возврата указателя в начало каталога на первую запись существует вызов:

***void rewindir(DIR \*dirptr);***

Чтобы получить имя текущего рабочего каталога существует функция:

***char \*getcwd(char \*name, size\_t size);***

Время в ***Linux*** отсчитывается в секундах, прошедшее с начала этой эпохи (***00:00:00 UTC, 1 Января 1970 года***). Для получения системного времени можно использовать следующие функции:

***#include <sys/time.h>***

***time\_t time (time\_t \*tt);***

***int gettimeofday(struct timeval \*tv, struct timezone \*tz);***

***struct timeval {***

***long tv\_sec; /\* секунды \*/***

***long tv\_usec; /\* микросекунды \*/***

***};***

***#include <time.h>***

***#include <stdio.h>***

***#include <stdlib.h>***

***int main(int argc, char \*argv[])***

***{***

***char outstr[200];***

***time\_t t;***

***struct tm \*tmp;***

***t = time(NULL);***

***tmp = localtime(&t);***

***if (tmp == NULL) { perror("localtime"); exit(EXIT\_FAILURE); }***

***if (strftime(outstr, sizeof(outstr), argv[1], tmp) == 0) { fprintf(stderr, "strftime returned 0");***

***exit(EXIT\_FAILURE);***

***}***

***printf("Result string is \"%s\"\n", outstr);***

***exit(EXIT\_SUCCESS);***

***}***

**Порядок выполнения работы**

1. Изучить теоретическую часть лабораторной работы.
2. Написать программу вывода сообщения на экран.
3. Написать программу ввода символов с клавиатуры и записи их в файл (в качестве аргумента при запуске программы вводится имя файла). Для чтения или записи файла использовать ***только*** функции посимвольного ввода-вывода ***getc(),putc()***, ***fgetc()***,***fputc()***. Предусмотреть выход после ввода определённого символа (например: ***ctrl-F***). Предусмотреть контроль ошибок открытия/закрытия/чтения файла.
4. Написать программу вывода содержимого текстового файла на экран (в качестве аргумента при запуске программы передаётся имя файла, второй аргумент (***N***) устанавливает вывод по группам строк (по ***N*** –строк) или сплошным текстом (***N***=0)). Для вывода очередной группы строк необходимо ожидать нажатия пользователем любой клавиши. Для чтения или записи файла использовать ***только*** функции посимвольного ввода-вывода ***getc(),putc()***, ***fgetc()***,***fputc()***. Предусмотреть контроль ошибок открытия/закрытия/чтения/записи файла.
5. Написать программу копирования одного файла в другой. В качестве параметров при вызове программы передаются имена первого и второго файлов. Для чтения или записи файла использовать ***только*** функции посимвольного ввода-вывода ***getc(),putc()***, ***fgetc()***,***fputc()***. Предусмотреть копирование прав доступа к файлу и контроль ошибок открытия/закрытия/чтения/записи файла.
6. Написать программу вывода на экран содержимого текущего и корневого каталогов. Предусмотреть контроль ошибок открытия/закрытия/чтения каталога.

## Варианты индивидуальных заданий

1. Отсортировать в заданном каталоге (аргумент 1 командной строки) и во всех его подкаталогах файлы по следующим критериям (аргумент 2 командной строки, задаётся в виде целого числа):1 – по размеру файла, 2 – по имени файла. Записать без сохранения структуры каталогов отсортированные файлы общим списком, в новый каталог (аргумент 3 командной строки). В связи с индексированием файлов в каталогах для файловых систем ext 2,3,4 перед запуском программы необходимо временно отключить опцию индексирования файловой системы следующим образом:

***sudo*** ***tune2fs –O ^dir\_index /dev/sdaXY***

Проверить результат, используя, ***ls -l –f.***

1. Найти в заданном каталоге (аргумент 1 командной строки) и всех его подкаталогах заданный файл (аргумент 2 командной строки). Вывести на консоль полный путь к файлу, размер, дату создания, права доступа, номер индексного дескриптора. Вывести также общее количество просмотренных каталогов и файлов.
2. Для заданного каталога (аргумент 1 командной строки) и всех его подкаталогов вывести в заданный файл (аргумент 2 командной строки) и на консоль полный путь, размер и дату создания, удовлетворяющих заданным условиям: 1 – размер файла находится в заданных пределах от ***N1*** до ***N2*** (***N1,N2*** задаются в аргументах командной строки), 2 – дата создания находится в заданных пределах от ***M1*** до ***M2*** (***M1***,***M2*** задаются в аргументах командной строки).
3. Найти совпадающие по содержимому файлы в двух заданных каталогах (аргументы 1 и 2 командной строки) и всех их подкаталогах. Вывести на консоль и в файл (аргумент 3 командной строки) полный путь, размер, дату создания, права доступа, номер индексного дескриптора.
4. Подсчитать суммарный размер файлов в заданном каталоге (аргумент 1 командной строки) и для каждого его подкаталога отдельно. Вывести на консоль и в файл (аргумент 2 командной строки) название подкаталога, количество файлов в нём, суммарный размер файлов, имя файла с наибольшим размером.
5. Написать программу, находящую в заданном каталоге и всех его подкаталогах все файлы, заданного размера. Имя каталога задаётся пользователем в качестве первого аргумента командной строки. Диапазон от ***N1*** до ***N2*** задается в аргументах командной строки. Программа выводит результаты поиска в файл (четвертый аргумент командной строки) в виде полный путь, имя файла, его размер. На консоль выводится общее число просмотренных файлов.
6. Найти все дубликаты (с одинаковым содержимым) файлов в заданном диапазоне размеров от ***N1*** до ***N2*** (***N1, N2*** задаются в аргументах командной строки), начиная с исходного каталога и ниже. Имя исходного каталога задаётся пользователем в качестве первого аргумента командной строки.
7. Подсчитать для заданного каталога (первый аргумент командной строки) и всех его подкаталогов (по-отдельности) суммарный размер занимаемого файлами на диске пространства в байтах и суммарный размер файлов. Вычислить коэффициент использования дискового пространства в %. Для получения размера занимаемого файлами на диске пространства использовать команду ***stat.***
8. Написать программу, находящую в заданном каталоге (первый аргумент командной строки) и всех его подкаталогах все файлы заданного расширения и создающий для каждого найденного файла жесткую ссылку в заданном каталоге. Расширение файла и каталог для жестких ссылок задаются в качестве второго и третьего аргументов командной строки.

# ЛАБОРАТОРНАЯ РАБОТА №3 ПРОЦЕССЫ В ОС LINUX

Цель работы – изучение вопросов порождения и взаимодействия процессов в ОС LINUX.

**Теоретическая часть**

В ОС Linux для создания процессов используется системный вызов ***fork()****:*

***#include <sys/types.h>***

***#include <unistd.h>***

***pid\_t fork (void);***

В результате успешного вызова ***fork()*** ядро создаёт новый процесс, который является почти точной копией вызывающего процесса. Другими словами, новый процесс выполняет копию той же программы, что и создавший его процесс, при этом все его объекты данных имеют те же самые значения, что и в вызывающем процессе. Созданный процесс называется ***дочерним процессом***, а процесс, осуществивший вызов ***fork()***, называется ***родительским***. После вызова родительский процесс и его вновь созданный потомок выполняются одновременно, при этом оба процесса продолжают выполнение с оператора, который следует сразу же за вызовом ***fork()***. Процессы выполняются в разных адресных пространствах, поэтому прямой доступ к переменным одного процесса из другого процесса невозможен.

Следующая короткая программа более наглядно показывает работу вызова ***fork()*** и использование процесса:

***#include <stdio.h>***

***#include <unistd.h>***

***int main ()***

***{***

***pid\_t pid; /\* идентификатор процесса \*/***

***printf (“Пока всего один процесс\n”);***

***pid = fork (); /\* Создание нового процесса \*/***

***printf (“Уже два процесса\n”);***

***if (pid = = 0){***

***printf (“Это Дочерний процесс его pid=%d\n”, getpid());***

***printf (“А pid его Родительского процесса=%d\n”, getppid());***

***}***

***else if (pid > 0)***

***printf (“Это Родительский процесс pid=%d\n”, getpid());***

***else***

***printf (“Ошибка вызова fork, потомок не создан\n”);***

***}***

Для корректного завершения дочернего процесса в родительском процессе необходимо использовать функцию ***wait()*** или ***waitpid()***:

***pid\_t wait(int \*status);   
pid\_t waitpid(pid\_t pid, int \*status, int options);***

Функция ***wait*** приостанавливает выполнение родительского процесса до тех пор, пока дочерний процесс не прекратит выполнение или до появления сигнала, который либо завершает текущий процесс, либо требует вызвать функцию-обработчик. Если дочерний процесс к моменту вызова функции уже завершился (так называемый ***«зомби»***), то функция немедленно возвращается. Системные ресурсы, связанные с дочерним процессом, освобождаются.

Функция ***waitpid ()*** приостанавливает выполнение родительского процесса до тех пор, пока дочерний процесс, указанный в параметре ***pid***, не завершит выполнение, или пока не появится сигнал, который либо завершает родительский процесс, либо требует вызвать функцию-обработчик. Если указанный дочерний процесс к моменту вызова функции уже завершился (так называемый ***«зомби»***), то функция немедленно возвращается. Системные ресурсы, связанные с дочерним процессом, освобождаются. Параметр ***pid*** может принимать несколько значений:

***pid< -1*** означает, что нужно ждать любого дочернего процесса, чей идент ификатор группы процессов равен абсолютному значению ***pid***.

***pid= -1*** означает ожидать любого дочернего процесса; функция **wait** ведет себя точно так же.

***pid = 0*** означает ожидать любого дочернего процесса, чей идентификатор группы процессов равен таковому у текущего процесса.

***pid> 0*** означает ожидать дочернего процесса, чем идентификатор равен ***pid***.

Значение ***options*** создается путем битовой операции ***ИЛИ*** над следующими константами:

**WNOHANG** - означает вернуть управление немедленно, если ни один дочерний процесс не завершил выполнение.

**WUNTRACED -** означает возвращать управление также для остановленных дочерних процессов, о чьем статусе еще не было сообщено.

Каждый дочерний процесс при завершении работы посылает своему процессу-родителю специальный сигнал **SIGCHLD**, на который у всех процессов по умолчанию установлена реакция "игнорировать сигнал". Наличие такого сигнала совместно с системным вызовом **waitpid()** позволяет организовать асинхронный сбор информации о статусе завершившихся порожденных процессов процессом-родителем.

Для перегрузки исполняемой программы можно использовать функции семейства ***exec***. Основное отличие между разными функциями в семействе состоит в способе передачи параметров.

***int execl(char \*pathname, char \*arg0, arg1, ..., argn, NULL);***

***int execle(char \*pathname, char \*arg0, arg1, ..., argn, NULL, char \*\*envp);***

***int execlp(char \*pathname, char \*arg0, arg1, ..., argn, NULL);***

***int execlpe(char \*pathname, char \*arg0, arg1, ..., argn, NULL, char \*\*envp);***

***int execv(char \*pathname, char \*argv[]);***

***int execve(char \*pathname, char \*argv[],char \*\*envp);***

***int execvp(char \*pathname, char \*argv[]);***

***int execvpe(char \*pathname, char \*argv[],char \*\*envp);***

Основное отличие между разными функциями в семействе состоит в способе передачи параметров. Как видно из рис. 1, все эти функции выполняют один системный вызов ***execve****.*

**execl**

**execle**

**execve**

**execvp**

**execv**

**execlp**

Рис. 1. Дерево семейства вызовов ***exec***

**Порядок выполнения работы**

* Изучить теоретическую часть лабораторной работы.
* Написать программу, создающую два дочерних процесса с использованием двух вызовов ***fork()***. Родительский и два дочерних процесса должны выводить на экран свой ***pid*** и ***pid*** родительского процесса и текущее время в формате: ***часы: минуты: секунды: миллисекунды***. Используя вызов ***system (),*** выполнить команду ***ps -x*** в родительском процессе. Найти свои процессы в списке запущенных процессов.

## Варианты индивидуальных заданий

Все исходные данные ко всем заданиям вводятся как аргументы командной строки!

1. Написать программу нахождения массива значений функции ***y[i]=sin(2\*PI\*i/N)*** ***(i=0,1,2…N-1)*** с использованием ряда Тейлора. Пользователь задаёт значения ***N*** и количество ***n*** членов ряда Тейлора. Для расчета одного члена ряда Тейлора запускается отдельный процесс. Каждый процесс выводит на экран и в файл промежуточных результатов (создать в каталоге ***/tmp)*** свой ***pid***, ***i*** и рассчитанное значение члена ряда. Головной процесс считывает из файла промежуточных результатов значения всех рассчитанных членов ряда Тейлора для каждого ***i***, суммирует их и полученное значение ***y[i]*** записывает в файл результата в виде: ***y[i] =значение***. Проверить работу программы для ***N=256 n=5; N=1024 n=10.***
2. Написать программу синхронизации двух каталогов, например, ***Dir1*** и ***Dir2***. Пользователь задаёт имена ***Dir1*** и ***Dir2***. В результате работы программы файлы, имеющиеся в ***Dir1***, но отсутствующие в ***Dir2***, должны скопироваться в ***Dir2*** вместе с правами доступа. Процедуры копирования должны запускаться в отдельном процессе для каждого копируемого файла. Каждый процесс выводит на экран свой ***pid,*** полный путь к копируемому файлу и число скопированных байт. Число одновременно работающих процессов не должно превышать ***N*** (вводится пользователем). Скопировать несколько файлов из каталога ***/etc*** в свой домашний каталог. Проверить работу программы для каталога ***/etc и*** домашнего каталога.
3. Написать программу поиска одинаковых по содержимому файлов в двух каталогов, например, ***Dir1*** и ***Dir2***. Пользователь задаёт имена ***Dir1*** и ***Dir2***. В результате работы программы файлы, имеющиеся в ***Dir1***, сравниваются с файлами в ***Dir2*** по их содержимому. Процедуры сравнения должны запускаться в отдельном процессе для каждой пары сравниваемых файлов. Каждый процесс выводит на экран свой ***pid***, имя файла, общее число просмотренных байт и результаты сравнения. Число одновременно работающих процессов не должно превышать ***N*** (вводится пользователем). Скопировать несколько файлов из каталога ***/etc*** в свой домашний каталог. Проверить работу программы для каталога ***/etc и*** домашнего каталога.
4. Написать программу поиска заданной пользователем комбинации из ***m*** байт (***m <255***) во всех файлах заданного каталога. Главный процесс открывает каталог и запускает для каждого файла каталога отдельный процесс поиска заданной комбинации из ***m*** байт. Каждый процесс выводит на экран свой ***pid,*** полный путь к файлу, общее число просмотренных байт и результаты (сколько раз найдена комбинация) поиска (все в одной строке). Число одновременно работающих процессов не должно превышать ***N*** (вводится пользователем). Проверить работу программы для каталога ***/etc и*** строки ***«ifconfig»***.
5. Написать программу подсчета количества слов в файлах заданного каталога его подкаталогов. Пользователь задаёт имя каталога. Главный процесс открывает каталоги и запускает для каждого файла каталога отдельный процесс подсчета количества слов. Каждый процесс выводит на экран свой ***pid,*** полный путь к файлу, общее число просмотренных байт и количество слов. Число одновременно работающих процессов не должно превышать ***N*** (вводится пользователем). Проверить работу программы для каталога ***/etc***.
6. Написать программу подсчета частоты встречающихся символов в файлах заданного каталога его подкаталогов. Пользователь задаёт имя каталога. Главный процесс открывает каталоги и запускает для каждого файла каталога и отдельный процесс подсчета количества символов. Каждый процесс выводит на экран свой ***pid,*** полный путь к файлу, общее число просмотренных байт и частоты встречающихся символов (все в одной строке). Число одновременно работающих процессов не должно превышать ***N*** (вводится пользователем). Проверить работу программы для каталога ***/etc***.
7. Написать программу подсчета количества бит (0 и 1) в файлах заданного каталога и его подкаталогов. Пользователь задаёт имя каталога. Главный процесс открывает каталоги и запускает для каждого файла каталога и отдельный процесс подсчета количества бит. Каждый процесс выводит на экран свой ***pid,*** полный путь к файлу, общее число просмотренных байт и количество бит 0 и 1. Число одновременно работающих процессов не должно превышать ***N*** (вводится пользователем). Проверить работу программы для каталога ***/etc***.
8. Написать программу подсчета всех периодов бит (0 и 1) в файлах заданного каталога и его подкаталогов. Пользователь задаёт имя каталога. Главный процесс открывает каталоги и запускает для каждого файла каталога и отдельный процесс подсчета всех периодов бит. Каждый процесс выводит на экран свой ***pid,*** полный путь к файлу, общее число просмотренных байт и все периоды бит (0 и 1). Число одновременно работающих процессов не должно превышать ***N*** (вводится пользователем). Проверить работу программы для каталога ***/etc***.
9. Написать программу шифрации всех файлов заданного каталога и его подкаталогов. Пользователь задаёт имя каталога. Главный процесс открывает каталоги и запускает для каждого файла каталога и отдельный процесс шифрации. Каждый процесс выводит на экран свой ***pid,*** полный путь к файлу, общее число зашифрованных байт. Шифрация по алгоритму сложения по модулю 2 бит исходного файла и файла ключа. Число одновременно работающих процессов не должно превышать ***N*** (вводится пользователем). Проверить работу программы для каталога ***/etc***.

# *ЛАБОРАТОРНАЯ РАБОТА №4* ИСПОЛЬЗОВАНИЕ СИГНАЛОВ В ОС LINUX

Цель работы – изучение механизма взаимодействия процессов с использованием сигналов.

**Теоретическая часть**

Сигналы не могут непосредственно переносить информацию, что ограничивает их применимость в качестве общего механизма межпроцессного взаимодействия. Тем не менее, каждому типу сигналов присвоено мнемоническое имя (например, ***SIGINT***), которое указывает, для чего обычно используется сигнал этого типа. Имена сигналов определены в стандартном заголовочном файле ***<signal.h>*** при помощи директивы препроцессора ***#define***. Как и следовало ожидать, эти имена соответствуют небольшим положительным целым числам. С точки зрения пользователя получение процессом сигнала выглядит как возникновение прерывания. Процесс прерывает исполнение, и управление передается функции-обработчику сигнала. По окончании обработки сигнала процесс может возобновить регулярное исполнение. Типы сигналов принято задавать специальными символьными константами. Системный вызов ***kill()*** предназначен для передачи сигнала одному или нескольким специфицированным процессам в рамках полномочий пользователя.

***#include <sys/types.h>***

***#include <signal.h>***

***int kill(pid\_t pid, int signal);***

Послать сигнал (не имея полномочий суперпользователя) можно только процессу, у которого эффективный идентификатор пользователя совпадает с эффективным идентификатором пользователя для процесса, посылающего сигнал. Аргумент ***pid*** указывает процесс, которому посылается сигнал, а аргумент ***sig*** – какой сигнал посылается. В зависимости от значения аргументов:

***pid > 0*** сигнал посылается процессу с идентификатором ***pid***;

***pid=0*** сигнал посылается всем процессам в группе, к которой принадлежит посылающий процесс;

***pid=-1*** и посылающий процесс не является процессом суперпользователя, то сигнал посылается всем процессам в системе, для которых идентификатор пользователя совпадает с эффективным идентификатором пользователя процесса, посылающего сигнал.

***pid = -1*** и посылающий процесс является процессом суперпользователя, то сигнал посылается всем процессам в системе, за исключением системных процессов (обычно всем, кроме процессов ***с pid = 0*** и ***pid = 1***).

***pid < 0***, но не ***–1***, то сигнал посылается всем процессам из группы, идентификатор которой равен абсолютному значению аргумента ***pid*** (если позволяют привилегии).

если ***sig = 0***, то производится проверка на ошибку, а сигнал не посылается. Это можно использовать для проверки правильности аргумента ***pid*** (есть ли в системе процесс или группа процессов с соответствующим идентификатором).

Системные вызовы для установки собственного обработчика сигналов:

***#include <signal.h>***

***void (\*signal (int sig, void (\*handler) (int)))(int);***

***int sigaction(int sig, const struct sigaction \*act, struct sigaction \*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);***

Системный вызов ***signal*** служит для изменения реакции процесса на какой-либо сигнал. Параметр ***sig*** – это номер сигнала, обработку которого предстоит изменить. Параметр handler описывает новый способ обработки сигнала – это может быть указатель на пользовательскую функцию-обработчик сигнала, специальное значение ***SIG\_DFL*** (восстановить реакцию процесса на сигнал ***sig*** по умолчанию) или специальное значение ***SIG\_IGN*** (игнорировать поступивший сигнал ***sig***). Системный вызов возвращает указатель на старый способ обработки сигнала, значение которого можно использовать для восстановления старого способа в случае необходимости.

Пример пользовательской обработки сигнала ***SIGUSR1***.

***void \*my\_handler(int nsig) {*** код функции-обработчика сигнала ***}***

***int main() {***

***(void) signal(SIGUSR1, my\_handler); }***

Системный вызов ***sigaction*** используется для изменения действий процесса при получении соответствующего сигнала. Параметр ***sig*** задает номер сигнала и может быть равен любому номеру. Если параметр ***act*** не равен нулю, то новое действие, связянное с сигналом ***sig***, устанавливается соответственно ***act***. Если ***oldact*** не равен нулю, то предыдущее действие записывается в ***oldact***.

Большинство типов сигналов ***UNIX*** предназначены для использования ядром, хотя есть несколько сигналов, которые посылаются от процесса к процессу:

***SIGALRM*** – сигнал таймера (***alarm clock***). Посылается процессу ядром при срабатывании таймера. Каждый процесс может устанавливать не менее трех таймеров. Первый из них измеряет прошедшее реальное время. Этот таймер устанавливается самим процессом при помощи системного вызова ***alarm()***;

***SIGCHLD*** – сигнал останова или завершения дочернего процесса (***child process terminated or stopped***). Если дочерний процесс останавливается или завершается, то ядро сообщит об этом родительскому процессу, послав ему данный сигнал. По умолчанию родительский процесс игнорирует этот сигнал, поэтому, если в родительском процессе необходимо получать сведения о завершении дочерних процессов, то нужно перехватывать этот сигнал;

***SIGHUP*** – сигнал освобождения линии (***hangup signal***). Посылается ядром всем процессам, подключенным к управляющему терминалу (***control terminal***) при отключении терминала. Он также посылается всем членам сеанса, если завершает работу лидер сеанса (обычно процесс командного интерпретатора), связанного с управляющим терминалом;

***SIGINT*** – сигнал прерывания программы (***interrupt***). Посылается ядром всем процессам сеанса, связанного с терминалом, когда пользователь нажимает клавишу прерывания. Это также обычный способ остановки выполняющейся программы;

***SIGKILL*** – сигнал уничтожения процесса (***kill***). Это довольно специфический сигнал, который посылается от одного процесса к другому и приводит к немедленному прекращению работы получающего сигнал процесса;

***SIGPIPE*** – сигнал о попытке записи в канал или сокет, для которых принимающий процесс уже завершил;

***SIGPOLL*** – сигнал о возникновении одного из опрашиваемых событий (***pollable event***). Этот сигнал генерируется ядром, когда некоторый открытый дескриптор файла становится готовым для ввода или вывода;

***SIGPROF*** – сигнал профилирующего таймера (***profiling time expired***). Как было упомянуто для сигнала ***SIGALRM***, любой процесс может установить не менее трех таймеров. Второй из этих таймеров может использоваться для измерения времени выполнения процесса в пользовательском и системном режимах. Этот сигнал генерируется, когда истекает время, установленное в этом таймере, и поэтому может быть использован средством профилирования программы;

***SIGQUIT*** – сигнал о выходе (***quit***). Очень похожий на сигнал ***SIGINT***, этот сигнал посылается ядром, когда пользователь нажимает клавишу выхода используемого терминала. В отличие от ***SIGINT***, этот сигнал приводит к аварийному завершению и сбросу образа памяти;

***SIGSTOP*** – сигнал останова (***stop executing***). Это сигнал управления заданиями, который останавливает процесс. Его, как и сигнал ***SIGKILL***, нельзя проигнорировать или перехватить;

***SIGTERM*** – программный сигнал завершения (***software termination signal***). Программист может использовать этот сигнал для того, чтобы дать процессу время для «наведения порядка», прежде чем посылать ему сигнал ***SIGKILL***;

***SIGTRAP*** – сигнал трассировочного прерывания (***trace trap***). Это особый сигнал, который в сочетании с системным вызовом ptrace используется отладчиками, такими как ***sdb, adb, gdb***;

***SIGTSTP*** – терминальный сигнал остановки (***terminal stop signal***). Он формируется при нажатии специальной клавиши останова;

***SIGTTIN*** – сигнал о попытке ввода с терминала фоновым процессом (***background process attempting read***). Если процесс выполняется в фоновом режиме и пытается выполнить чтение с управляющего терминала, то ему посылается этот сигнал. Действие сигнала по умолчанию – остановка процесса;

***SIGTTOU*** – сигнал о попытке вывода на терминал фоновым процессом (***background process attempting write***). Аналогичен сигналу ***SIGTTIN***, но генерируется, если фоновый процесс пытается выполнить запись в управляющий терминал. Действие сигнала по умолчанию – остановка процесса;

***SIGURG*** – сигнал о поступлении в буфер сокета срочных данных (***high bandwidth data is available at a socket***). Он сообщает процессу, что по сетевому соединению получены срочные внеочередные данные;

***SIGUSR1*** и ***SIGUSR2*** – пользовательские сигналы (***user defined signals 1 and 2***). Так же, как и сигнал ***SIGTERM***, эти сигналы никогда не посылаются ядром и могут использоваться для любых целей по выбору пользователя;

***SIGVTALRM*** – сигнал виртуального таймера (***virtual timer expired***). Третий таймер можно установить так, чтобы он измерял время, которое процесс выполняет в пользовательском режиме.

Наборы сигналов определяются при помощи типа ***sigset\_t***, который определен в заголовочном файле ***<signal.h>***. Выбрать определенные сигналы можно, начав либо с полного набора сигналов и удалив ненужные сигналы, либо с пустого набора, включив в него нужные. Инициализация пустого и полного набора сигналов выполняется при помощи процедур ***sigemptyset*** и ***sigfillset*** соответственно. После инициализации с наборами сигналов можно оперировать при помощи процедур ***sigaddset*** и ***sigdelset***, соответственно добавляющих и удаляющих указанные вами сигналы.

Описание данных процедур:

***#include <signal.h>***

/\* Инициализация\*/

***int sigemptyset (sigset\_t \*set);***

***int sigfillset (sigset\_t \*set);***

/\*Добавление и удаление сигналов\*/

***int sigaddset (sigset\_t \*set, int signo);***

***int sigdelset (sigset\_t \*set, int signo);***

Процедуры ***sigemptyset*** и ***sigfillset*** имеют единственный параметр – указатель на переменную типа ***sigset\_t***. Вызов ***sigemptyset*** инициализирует набор ***set***, исключив из него все сигналы. И, наоборот, вызов ***sigfillset*** инициализирует набор, на который указывает ***set***, включив в него все сигналы. Приложения должны вызывать ***sigemptyset*** или ***sigfillset*** хотя бы один раз для каждой переменной типа ***sigset\_t***.

Процедуры ***sigaddset*** и ***sigdelset*** принимают в качестве параметров указатель на инициализированный набор сигналов и номер сигнала, который должен быть добавлен или удален. Второй параметр, ***signo***, может быть символическим именем константы, таким как ***SIGINT***, или настоящим номером сигнала, но в последнем случае программа окажется системно-зависимой.

**Порядок выполнения работы**

1. Изучить теоретическую часть лабораторной работы.
2. Организовать функционирование процессов следующей структуры:

Процессы определяют свою работу выводом сообщений вида :

***N pid ppid текущее время*** (мсек) (***N*** – текущий номер сообщения) на экран. “Отец” одновременно, посылает сигнал ***SIGUSR1*** “сыновьям”. “Сыновья” получив данный сигнал, посылают в ответ “Отцу” сигнал ***SIGUSR2***. “Отец” получив сигнал ***SIGUSR2***, через время ***t=100*** мсек одновременно, посылает сигнал ***SIGUSR1*** “сыновьям”. И так далее… Написать функции-обработчики сигналов, которые при получении сигнала выводят сообщение о получении сигнала на экран. При получении/посылке сигнала они выводят соответствующее сообщение:

***N pid ppid текущее время*** (мсек) ***сын такой-то get/put SIGUSRm***.

Предусмотреть механизм для определения “Отцом”, от кого из “Сыновей” получен сигнал.

## Варианты индивидуальных заданий

Создать дерево процессов согласно варианта индивидуального задания.

Процессы непрерывно обмениваются сигналами согласно табл. 2 . Запись в таблице 1 вида: ***1->(2,3,4,5)*** означает, что исходный процесс ***0*** создаёт дочерний процесс ***1,*** который, в свою очередь, создаёт дочерние процессы ***2,3,4,5.*** Запись в таблице 2 вида: ***1->(2,3,4) SIGUSR1*** означает, что процесс 1 посылает дочерним процессам ***2,3,4*** одновременно (т.е. за один вызов ***kill()*** ) сигнал ***SIGUSR1***.Каждый процесс при получении или посылке сигнала выводит на консоль информацию в следующем виде:

***N pid ppid послал/получил USR1/USR2 текущее время (мксек)***

где ***N***-номер сына по табл. 1

Процесс 1, после получения ***101*** –го по счету сигнала ***SIGUSR,*** посылает сыновьям сигнал ***SIGTERM*** и ожидает завершения всех сыновей, после чего завершается сам***.*** Процесс 0 ожидает завершения работы процесса 1 после чего завершается сам. Сыновья, получив сигнал ***SIGTERM,*** завершают работу с выводом на консоль сообщения вида:

***pid ppid завершил работу после X-го сигнала SIGUSR1 и Y-го сигнала SIGUSR2***

где ***X, Y*** – количество посланных за все время работы данным сыном сигналов ***SIGUSR1*** и ***SIGUSR2***

***Для создания правильной последовательности сигналов в соответствие с таблицей задания необходимо для каждого процесса написать свой обработчик сигналов в котором он (процесс) принимает сигнал от предыдущего (в таблице) процесса и посылает следующему (в таблице) процессу!!***

**Во всех заданиях** д**олжен быть контроль ошибок (**если к какому-либо каталогу нет доступа, необходимо вывести соответствующее сообщение и продолжить выполнение).

Вывод сообщений об ошибках должен производиться в стандартный поток вывода сообщений об ошибках (***stderr***) в следующем виде:

***имя\_модуля: текст\_сообщения***.

Пример: pid: 1.exe***: Error open file: 1.txt***

Имя модуля, имя файла берутся из аргументов командной строки.

Варианты индивидуальных заданий в табл.1, табл.2.

**Дерево процессов**

Таблица 1

|  |  |
| --- | --- |
| **№** | **Дерево процессов** |
| **1** | **1->2 2->(3,4) 4->5 3->6 6->7 7->8** |
| **2** | **1->(2,3,4) 2->(5,6) 6->7 7->8** |
| **3** | **1->(2,3,4,5) 2->6 3->7 4->8** |
| **4** | **1->(2,3) 2->(4,5) 5->6 6->(7,8)** |
| **5** | **1->(2,3,4,5) 5->(6,7,8)** |
| **6** | **1->(2,3) 3->4 4->(5,6,7) 7->8** |
| **7** | **1->2 2->(3,4) 4->5 3->6 6->7 7->8** |
| **8** | **1->(2,3,4,5,6) 6->(7,8)** |
| **9** | **1->2 2->(3,4,5) 4->6 3->7 5->8** |
| **10** | **1->2 2->3 3->(4,5,6) 6->7 4->8** |
| **11** | **1->(2,3) 3->4 4->(5,6) 6-7 7->8** |
| **12** | **1->2 2->(3,4) 4->5 3->6 6->7 7->8** |
| **13** | **1->(2,3,4,5) 5->(6,7) 7->8** |
| **14** | **1->2 2->(3,4,5) 4->6 3->7 5->8** |
| **15** | **1->2 2->3 3->(4,5,6) 6->7 4->8** |
| **16** | **1->(2,3,4,5) 2->(6,7) 7->8** |

**Последовательность обмена сигналами**

Таблица 2

|  |  |
| --- | --- |
| **№** | **Последовательность обмена сигналами** |
| **1** | ***1->2 SIGUSR1 2->(3,4) SIGUSR2 4->5 SIGUSR1***  ***3->6 SIGUSR1 6->7 SIGUSR1 7->8 SIGUSR2 8->1 SIGUSR2*** |
| **2** | ***1->(2,3,4) SIGUSR1 2->(5,6) SIGUSR2 6->7 SIGUSR1***  ***7->8 SIGUSR1 8->1 SIGUSR2*** |
| **3** | ***1->(2,3,4,5) SIGUSR2 2->6 SIGUSR1 3->7 SIGUSR1 4->8 SIGUSR1 8->1 SIGUSR1*** |
| **4** | ***1->(2,3) SIGUSR1 2->(4,5) SIGUSR1 5->6 SIGUSR1***  ***6->(7,8) SIGUSR1 8->1 SIGUSR1*** |
| **5** | ***1->(2,3,4,5) SIGUSR1 5->(6,7,8) SIGUSR1 8->1 SIGUSR1*** |
| **6** | ***1->(2,3) SIGUSR1 3->4 SIGUSR2 4->(5,6,7) SIGUSR1***  ***7->8 SIGUSR1 8->1 SIGUSR2*** |
| **7** | ***1->2 SIGUSR1 2->(3,4) SIGUSR2 4->5 SIGUSR1***  ***3->6 SIGUSR1 6->7 SIGUSR1 7->8 SIGUSR1 8->1 SIGUSR1*** |
| **8** | ***1->(2,3,4,5,6) SIGUSR2 6->(7,8) SIGUSR1 8->1 SIGUSR2*** |
| **9** | ***1->2 SIGUSR2 2->(3,4,5) SIGUSR1 4->6 SIGUSR1***  ***3->7 SIGUSR1 5->8 SIGUSR1 8->1 SIGUSR2*** |
| **10** | **1->(8,7,6)** ***SIGUSR1*  8->4 *SIGUSR1* 7->4*SIGUSR2***  **6->4** ***SIGUSR1* 4->(3,2) *SIGUSR1* 2->1** ***SIGUSR2*** |
| **11** | **1->(8,7)** ***SIGUSR1*  8->(6,5) *SIGUSR1* 5->(4,3,2)** ***SIGUSR2***  **2->1** ***SIGUSR2*** |
| **12** | **1->(8,7,6,5)** ***SIGUSR1*  8->3 *SIGUSR1* 7->3** ***SIGUSR2***  **6->3** ***SIGUSR1* 5->3 *SIGUSR1* 3->2*SIGUSR2* 2->1 *SIGUSR2*** |
| **13** | **1->6** ***SIGUSR1*  6->7 *SIGUSR1* 7->(4,5)** ***SIGUSR2***  **4->8** ***SIGUSR1* 5->2 *SIGUSR1* 8->2** ***SIGUSR2* 2->1** ***SIGUSR2*** |
| **14** | **1->8** ***SIGUSR1*  8->7 *SIGUSR1* 7->(4,5,6)** ***SIGUSR2***  **4->2** ***SIGUSR1* 2->3 *SIGUSR1* 3->1** ***SIGUSR2*** |