ФЕДЕРАЛЬНОЕ АГЕНТСТВО ВОЗДУШНОГО ТРАНСПОРТА

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ ГРАЖДАНСКОЙ АВИАЦИИ» (МГТУ ГА)

Кафедра вычислительных машин, комплексов, систем и сетей

Н.И. Романчева

# СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ.

## ОС LINUX: УПРАВЛЕНИЕ ФАЙЛАМИ, ПРОЦЕССАМИ И СЕРВИСАМИ

**Учебно-методическое пособие**
по выполнению лабораторных работ №1–2

*для студентов*
*направления 09.03.01*
*очной формы обучения*

Москва
ИД Академии Жуковского
2020

УДК 004.451
ББК 6Ф7.3
Р69

Рецензент:
*Терентьев А.И.* – канд. техн. наук, доцент

**Романчева Н.И.**

Р69 Системное программное обеспечение. ОС LINUX: управление файлами, процессами и сервисами [Текст] : учебно-методическое пособие по выполнению лабораторных работ №1–2 / Н.И. Романчева. – М.: ИД Академии Жуковского, 2020. – 24 с.

Данное учебно-методическое пособие издается в соответствии с рабочей программой учебной дисциплины «Системное программное обеспечение» по учебному плану для студентов направления 09.03.01 очной формы обучения.

Рассмотрено и одобрено на заседаниях кафедры 28.01.2020 г. и методического совета 28.01.2020 г.

**УДК 004.451**
**ББК 6Ф7.3**

*В авторской редакции*

Подписано в печать 16.06.2020 г.
Формат 60x84/16 Печ. л. 1,5 Усл. печ. л. 1,395
Заказ № 565/0225-УМП25 Тираж 35 экз.

Московский государственный технический университет ГА
125993, Москва, Кронштадтский бульвар, д. 20

Издательский дом Академии имени Н. Е. Жуковского
125167, Москва, 8-го Марта 4-я ул., д. 6А
Тел.: (495) 973-45-68
E-mail: zakaz@itsbook.ru

## С О Д Е Р Ж А Н И Е

# 1. ОСНОВНЫЕ ТРЕБОВАНИЯ И ПОРЯДОК ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ

Настоящее учебно-методическое пособие предназначено для студентов направления подготовки Информатика и вычислительная техника (бакалавриат), выполняющих лабораторные работы по дисциплине Системное программное обеспечение. В данное учебно-методическое пособие включены материалы для выполнения лабораторных работ № 1-2.

Продолжительность каждой лабораторной работы - 4 часа.

Целью проведения лабораторных работ является закрепление основных теоретических положений, изложенных в лекциях по дисциплине Системное программное обеспечение, на примере ОС UNIX/Linux (Red Hat), широко используемых в различных областях.

В процессе выполнения лабораторных работ студенты должны получить практические навыки работы в ОС UNIX/Linux (Red Hat), в том числе: разработать, отладить и запустить программу со сложными синтаксическими конструкциями; изучение расширенных возможностей командных интерпретаторов;

В процессе выполнения лабораторной работы студенты должны освоить приемы и методы работы в ОС UNIX/Linux.

Лабораторная работа состоит из следующих этапов:

1) домашняя подготовка;

2) выполнение работы на компьютере в соответствии с заданием;

3) сдача выполненной работы преподавателю на персональном компьютере;

4) распечатка результатов работы на принтере;

5) оформление отчета;

6) защита лабораторной работы.

В процессе домашней подготовки студент:

- изучает лекционный материал,
- материал по теме лабораторной работы данного учебно-методического пособия и дополнительной литературы;
- знакомится с заданием на выполнение лабораторной работы;
- готовит проект отчета по выполнению лабораторной работы.

ВЫПОЛНЕНИЕ лабораторной работы проводится во время занятий в компьютерном классе МГТУГА в присутствии преподавателя в соответствии с расписанием, утвержденным проректором по УМР. В процессе выполнения лабораторной работы студент последовательно выполняет задание. По завершению работы - демонстрирует преподавателю результаты.

СДАЧА РАБОТЫ преподавателю на персональном компьютере заключается в демонстрации выполненной работы и выполнении непосредственно при преподавателе индивидуального дополнительного задания.

ОТЧЕТ по каждой лабораторной работе должен содержать: название работы; цель лабораторной работы; задание на выполнение лабораторной работы; краткие комментарии по выполнению лабораторной работы; распечатки файлов результатов, подписанные преподавателем.

ЗАЩИТА лабораторной работы преподавателю проводится по контрольным вопросам и при наличии оформленного отчета (распечатки результатов выполнения лабораторной работы должны быть приклеены). После защиты лабораторной работы преподавателем делается соответствующая запись на отчете студента.

РЕЙТИНГОВЫЙ КОНТРОЛЬ по лабораторным работам производится при их сдаче во время лабораторных занятий. Перед выполнением лабораторной работы производится экспресс-опрос для определения готовности студентов к выполнению работы (знания теоретического материала, целей работы и т.д.).

1) Допуск к ЛР

Допуск к выполнению ЛР происходит в виде устного опроса (при условии наличия у студента печатной версии титульного листа отчета по лабораторной работе) в форме тестирования (список из 2-4 тестовых вопросов выдается на занятии, время на ответ – 10 минут). Баллы начисляются в зависимости от количества правильных ответов: от 1 до 2 правильных ответов – min балл, более 3 правильных ответов – max балл.

Студент, не прошедший устный опрос к выполнению лабораторной работы не допускается.

2) Выполнение и защита ЛР

Минимальная оценка выставляется за выполненную и защищенную лабораторную работу, а дополнительными баллами оценивается качество выполненной лабораторной работы и полнота знаний, показанная студентами при ее защите. Лабораторная работа считается сделанной, если она выполнена полностью в соответствии с заданием. За сделанную не в сроки, устанавливаемые расписанием, лабораторную работу студент может получить только минимальное количество баллов.

*Шкала рейтингового контроля по лабораторным работам*

| Оценка за сделанную лабораторную работу | Количество баллов |
|---|---|
| удовлетворительно | 0,5 балла |
| хорошо | 1 балла |
| отлично | 1,5 баллов |

*Сроки сдачи лабораторных работ*

| Номер лабораторной работы | Лабораторная работа | Срок сдачи* |
|---|---|---|
| 1 | Лабораторная работа №1 | Лабораторная работа №1 |
| 2 | Лабораторная работа №2 | Лабораторная работа №2 |

**Срок сдачи лабораторной работы – в соответствии с расписанием проведения лабораторных работ.*

3) Отчет по ЛР

Отчет по лабораторной работе представляется в печатном виде в формате, предусмотренном шаблоном отчета по лабораторной работе.

4) Защита ЛР

Основаниями для снижения количества баллов в диапазоне от 1,2 до 0 являются: небрежное выполнение, низкое качество представленного материала (неполное решение, не указаны единицы измерения, отсутствуют выводы по работе).

Отчет не может быть принят и подлежит доработке в случае:

- отсутствия необходимых разделов отчета,
- отсутствия необходимого графического материала,
- некорректной обработки результатов выполнения лабораторной работы, и т.п.

Рейтинговый контроль по лабораторным работам

| Семестр | Количество лабораторных работ | Минимальное количество баллов за лабораторную работу | Максимальное количество баллов за лабораторную | Минимальное количество баллов за все лаб.раб. | Максимальное количество баллов за все лаб.раб |
|---|---|---|---|---|---|
| 5 | 4 | 1,5 | 2,25 | 6 | 9 |

## 2. ЛАБОРАТОРНАЯ РАБОТА №1
## ОС LINUX (RED HAT LINUX): УПРАВЛЕНИЕ ФАЙЛАМИ И ПРОЦЕССАМИ. РАСШИРЕННЫЕ ВОЗМОЖНОСТИ КОМАНДНЫХ ИНТЕРПРЕТАТОРОВ

### 2.1. Цель работы

Целью данной работы является изучение расширенных возможностей командных интерпретаторов, получение навыков отладки сценариев управление файлами и процессами в ОС семейства UNIX\LINUX.

### 2.2. Перечень компетенций, формируемых в ходе выполнения лабораторной работы

Профессиональная компетенция ПК-1:

ПК-1.1.3

Умеет разработать, отладить и запустить программу со сложными синтаксическими конструкциями; разработать, отладить программу, содержащую функции, возвращающие значения реального и исполнительного кодов идентификации пользователей процесса

ПК-1.3.2

Владеет навыками установки и выполнения начального конфигурирования ОС Linux.

### 2.3. Задание на выполнение работы

1) Написать и отладить программу, демонстрирующую использование функции setuid (процесс может изменять значение кода идентификации пользователя, под которым он исполняется).
2) Написать и отладить программу, позволяющую вывести информацию об остановленных и фоновых заданиях:
   - полный список,
   - с указанными номерами заданий,
   - идентификатор группы процессов и рабочий каталог;
   - только идентификатор группы процессов.
3) Написать и отладить программу, позволяющую выполнить поиск файлов по следующим критериям:
   - по размеру,
   - с именем core (образ процесса, создаваемый при неудачном его завершении и используемый в целях отладки).
4) Написать и отладить программу на языке интерпретатора Borne:
   - используя команду echo;
   - используя команду управления форматированием вывода (printf);
   - обрабатывающую ввод из потока STDIN;
   - использующую несколько переменных в аргументе команды read,

   рассмотреть случаи:

- число переменных в списке равно числу аргументов, считанных из потока;
- число переменных в списке меньше числа аргументов, считанных из потока;
- вычисления длины окружности и площади круга (используя команду bc);

5) Запустить программу ведения журнала logger, вывод программы записать в файл file.log, сохраняя все предыдущие записи.
6) Выполнить мониторинг системы, используя команды *df, du* и *ulimit* для управления дисковым пространством в UNIX.

### 2.4. Основные теоретические сведения

#### 2.4.1. Программа setuid

Ядро связывает с процессом два кода идентификации пользователя, не зависящих от кода идентификации процесса: реальный (действительный) код идентификации пользователя и исполнительный код или *setuid* (от "set user ID" - установить код идентификации пользователя, под которым процесс будет исполняться).

Реальный код идентифицирует пользователя, несущего ответственность за выполняющийся процесс.

Исполнительный код используется для установки прав собственности на вновь создаваемые файлы, для проверки прав доступа к файлу и разрешения на посылку сигналов процессам через функцию *kill.*

Процессы могут изменять исполнительный код, запуская с помощью функции *exec* программу *setuid* или запуская функцию *setuid* в явном виде.

Программа *setuid* представляет собой исполняемый файл, имеющий в поле режима доступа установленный бит *setuid*. Когда процесс запускает программу *setuid* на выполнение, ядро записывает в поля, содержащие реальные коды идентификации, в таблице процессов и в пространстве процесса код идентификации владельца файла. Чтобы как-то различать эти поля, назовем одно из них, которое хранится в таблице процессов, сохраненным кодом идентификации пользователя. Рассмотрим пример, иллюстрирующий разницу в содержимом этих полей.

Синтаксис вызова системной функции *setuid*: *setuid (uid),* где uid - новый код идентификации пользователя.

В примере 1 приведен текст программы, демонстрирующей использование функции setuid. Предположим, что исполняемый файл, полученный в результате трансляции исходного текста программы, имеет владельца с именем "evm" (код идентификации 8319) и установленный бит setuid; право его исполнения предоставлено всем пользователям. Допустим также, что пользователи "vmk" (код идентификации 5088) и "evm" являются владельцами файлов с теми же именами, каждый из которых доступен только для чтения и только своему владельцу.

Пример 1.

```
#include <fcntl.h>
    main()
    {
        int uid,euid,fdvmk,fdevm;
/* получить реальный UID */
        uid = getuid();
/* получить исполнительный UID */
        euid = geteuid();
        printf ("uid %d euid %d\n",uid,euid);
        fdvmk = open ("vmk",O_RDONLY);
        fdevm = open ("evm",O_RDONLY);
        printf ("fvmk %d fdevm %d\n",fdvmk,fdevm);
        setuid (uid);
        printf ("after setuid (%d): uid %d euid %d\n",uid,  getuid (),geteuid ());
        fdvmk = open ("vmk",O_RDONLY);
        fdevm = open ("evm",O_RDONLY);
        printf("fdvmk %d fdevm %d\n",fdvmk,fdevm);
        setuid (euid);
        printf ("after setuid(%d): uid %d euid %d\n",euid,  getuid (),geteuid ());
    }
```

В результате выполнения данной программы пользователю "*mjb*" выводится следующая информация:

```
uid 5088 euid 8319
fdvmk -1 fdevm 3
after setuid(5088): uid 5088 euid 5088
fdvmk 4 fdevm -1
after setuid(8319): uid 5088 euid 8319
```

Системные функции *getuid* и *geteuid* возвращают значения реального и исполнительного кодов идентификации пользователей процесса, для пользователя "*vmk*" это, соответственно, 5088 и 8319. Поэтому процесс не может открыть файл "*vmk*" (ибо он имеет исполнительный код идентификации пользователя (8319), не разрешающий производить чтение файла), но может открыть файл "*evm*". После вызова функции *setuid*, в результате выполнения которой в поле исполнительного кода идентификации пользователя ("*vmk*") заносится значение реального кода идентификации, на печать выводятся значения и того, и другого кода идентификации пользователя "*vmk*": оба равны 5088. Теперь процесс может открыть файл "*vmk*", поскольку он исполняется под кодом идентификации пользователя, имеющего право на чтение из файла, но не может открыть файл "*evm*". Наконец, после занесения в поле исполнительного

кода идентификации значения, сохраненного функцией *setuid* (8319), на печать снова выводятся значения 5088 и 8319.

### 2.4.2. Программирование на языке командного интерпретатора

Интерпретатор (shell) UNIX представляет собой интерфейс командной строки. По своей функциональности он похож на интерпретатор COMMAND.COM системы MS DOS. Одной из задач интерпретатора является обеспечение безопасного и структурированного доступа к ядру UNIX, т.е. фактически это программный уровень, который предоставляет среду для ввода команд, обеспечивая тем самым взаимодействие между пользователем и ядром операционной системы. Кроме того, встроенный в интерпретатор мощный язык программирования используется для решения различных задач: от автоматизации повторяющихся команд до написания сложных интерактивных программ обработки данных, получения информации из небольших баз данных.

В среде UNIX существует много командных интерпретаторов, среди которых можно выделить такие как sh, tcsh, ksh, csh, bash, в Linux по умолчанию используется bash.

Ниже приводятся примеры нескольких способов написания простой программы вывода фразы "Hello, ASRLinux!" на языке интерпретатора Bourne.

Пример 2.

```
#!/bin/sh
# Первая программа "Hello, ASRLinux!",
# реализованная для интерпретатора Bourne
echo
echo   " Hello, ASRLinux! "
echo
exit   0
```

Пример 3.

```
#!/bin/sh
# Программа, "с командой printf",
# реализованная для интерпретатора Bourne
printf    "\n" с командой printf\n\n!"
exit   0
```

Пример 4.

```
#!/bin/sh
# Программа, "воспринимающая ввод с клавиатуры",
# реализованная для интерпретатора Bourne
echo
echo   -n   "Введите name:    "
read   name
echo
```

```
echo "Hello, ${name}!"
echo
exit 0
```

В примерах 2-4 первые строки содержат последовательность символов #!, сообщающую ОС, что за ней следует имя командного интерпретатора, в котором сценарий должен выполняться. В данном случае это интерпретатор Bourne, /bin/sh. Строки, начинающиеся с символа #, являются комментариями, и интерпретатор их игнорирует. Команда *echo* выводит свои аргументы в стандартный вывод (обычно это экран). Команда *exit* завершает работу программы и возвращает код выхода родительской программе (обычно это командный интерпретатор). Код завершения, равный 0, указывает, что программа нормально завершила работу. Код, отличный от 0, сообщает об ошибке. Если статус завершения не указан явно, то программа возвращает код завершения последней выполненной команды.

Простого вывода сообщений на экран недостаточно для решения задач, поэтому язык сценариев использует переменные. Все переменные в программах командного интерпретатора хранятся как строки. Кроме присвоения значений переменным и их использования в сценарии, командный интерпретатор предоставляется возможность обрабатывать ввод из потока STDIN. Для чтения пользовательского ввода используется команда *read.* Команда *read* в качестве аргументов может иметь несколько переменных (пример 4). В этом случае всякий пустой символ трактуется как символ разделитель между значениями, присваиваемыми разным переменным. Если число аргументов, прочитанных из потока ввода, меньше, чем число переменных в списке, оставшимся переменным не присваивается никаких значений. Если аргументов больше, чем переменных, то все оставшиеся значения присваиваются последней переменной.

Пример 5.

```
#!/bin/sh
echo
echo -n "Введите три числа, разделенные пробелами или символом табуляции: "
read var 1 var2 var3
echo
echo "The of var1 is: ${var1}"
echo "The of var2 is: ${var2}"
echo "The of var3 is: ${var3}"
echo
exit 0
```

Для операций с числами с плавающей запятой, а также обработки сложных выражений, где порядок операций изменен, применяется команда *bc* (пример 6).

Пример 6.

```
#!/bin/sh
# Данная программа вычисляет длину окружности
# и площади круга
# Переменной присваивается число p
pi="3.14159265"
# пользователю выводится сообщение и запрашивается радиус
# окружности
echo
echo  -n   "Введите радиус:  "
read   radius
# Вычисляемые значения сохраняются в переменных
cir=$[ echo   $radius*$pi   | bc ]
area= $[ echo  $radius^2*$pi   | bc ]
#Программа выводит результаты и завершает работу
printf   "\n\nThe circumference is:\t$cir\n"
printf   "The area is:\t\t$area\n\n"
exit  0
```

После ввода текста программы и сохранения, необходимо сделать файл выполняемым. Для этого используется команда: *chmod u+x hello.* Эта команда устанавливает права владельца на запуск файла *hello.* Для запуска файла наберите в командной строке: *. /hello* .

### 2.4.3 Управление заданиями

Для просмотра списка текущих заданий используется команда jobs. В этом списке приводится порядковый номер задания, который можно использовать в любой команде, связанной с управлением заданиями. Использование ключей: –p позволяет выводить только идентификаторы групп процессов, -l – только идентификаторы заданий и групп заданий, k - информацию о заданиях, состояние которых изменилось со времени последнего запроса.

Для запуска команды в фоновом режиме необходимо добавить к ее имени символ амперсанта, например, *top &* .

Команда *logger>>file.log* позволит запустить программу ведения журнала logger , вывод программы запишется в файл file.log, сохраняя все предыдущие записи. Если файл не существует, то он будет создан.

### 2.4.4. Мониторинг системы

Для управления дисковым пространством в UNIX используются команды *df, du* и *ulimit*:

**df [-ключ]** – команда определяет, сколько свободного дискового пространства и индексных дескрипторов доступно в разделе смонтированного диска.

По умолчанию команда используется без параметров и выводит объем свободного пространства, например:

*/ (/dev/hdb1 ): 260836 blocks 12034 files*
*/home (/dev/sda1 ): 260836 blocks 2104 files*

В первом столбце содержится точка монтирования данной файловой системы. Затем в круглых скобках следует имя смонтированного физического устройства (в UNIX все устройства являются файлами, даже сама файловая система). Следующий столбец отображает число свободных блоков размером по 512 байт. В последнем столбце выводится количество файлов, содержащихся на данном устройстве.

При использовании ключей:

**-k –** вывод данных осуществляется в блоках по 1024 байт, или в килобайтах. При этом данные выводятся в формате, принятом в системе BSD:

| Filesystem | 1024- blocks | Used | AvailableCapacity | Mounted on |
|---|---|---|---|---|
| */dev/hdb1* | *1112646* | *972611* | *140035* | *88%* / |
| */dev/sda1* | *961374* | *720104* | *241270* | *75%* */home* |

В первом столбце указано имя устройства, на котором расположена файловая система. Во втором столбце отображается размер файловой системы в блоках по 1 Кбайт. В третьем столбце выводится число используемых блоков, а в четвертом – число свободных блоков. В пятом столбце выводится процент использования диска. В последнем столбце указывается точка монтирования системы;

**-P –** информация отображается в формате, определенном в стандарте POSIX, который аналогичен формату, принятому в BSD;

**-t -** информация отображается в формате, который близок к стилю, используемому в SYSTEM V. Данные выводятся в блоках размером по 512 байт, кроме того, приводится информация, как о количестве блоков, так и о количестве индексных дескрипторов;

**-i -** предназначен для подсчета количества индексных дескрипторов (не поддерживается стандартом POSIX). Выводимая информация имеет следующий вид:

| Filesystem | Inodes | IUsed | IFree | %IUsed | Mounted on |
|---|---|---|---|---|---|
| */dev/hdb1* | *301056* | *93059* | *207997* | *31%* | / |
| */dev/sda1* | *260096* | *17280* | *242816* | *7%* | */home* |

В качестве параметров команде *df* можно передать имя файла или список имен файлов. В этом случае отображается информация только о тех файловых системах, которые содержат указанные файлы.

**du [- ключ]** - команда определяет какой объем диска занимает конкретный каталог. Вызов команды без параметров позволяет получить данные о текущем каталоге. Если в качестве параметра указать имя каталога, то будет отображена информация обо всех каталогах, расположенных в иерархии

ниже текущего. Если в качестве параметра указано имя файла, не являющееся каталогом, то не выводится никакой информации.

Команда *du* имеет четыре ключа:

**-k –** имеет то же значение, что и для команды *df* , при этом данные об использовании дискового пространства представляются в килобайтах;

**-a –** задает вывод данных всех перечисленных файлов. При этом полученный результат аналогичен результатам выполнения команды *ls – ls*;

**-s** – задает ограниченный вывод, только данные об указанном каталоге, например: *13500 /home/nata/bin,* где 13500 – размер каталога, выраженный в блоках по 512 байт;

**-x –** не выводятся данные о файлах, находящихся в других файловых системах. Таким образом проверяются данные, хранящиеся в указанном каталоге локального диска;

**ulimit –** выводит или устанавливает значение пределов, ограничивающих использование задачей системных ресурсов (времени процессора, памяти, дискового пространства);

### 2.5. Вопросы к защите лабораторной работы

1) Что понимается под термином “пользовательская среда UNIX”?
2) Напишите общий синтаксис скрипта.
3) Приведите примеры сложных синтаксических конструкций получения значений переменной.
4) Перечислите внутренние переменные shell, используемые в скриптах.
5) Каким образом процессы могут изменять исполнительный код пользователя?
6) Какие действия выполняет ядро, если процесс запускает программу *setuid* на выполнение?
7) Назовите системные функции, возвращающие значения реального и исполнительного кодов идентификации пользователей процесса.

### 2.6. Список рекомендуемой литературы

1) Робачевский А. Операционная система UNIX.-СПб.:BHV-Санкт-Петербург, 2010- 528 с.
2) Романчева Н.И. Системное программное обеспечение. Учебное пособие. – М.: МГТУ ГА, 2013.
3) Романчева Н.И. ОС Linux: принципы, технологии, инструменты: Учебное пособие. – М.: МГТУ ГА, 2011.

# 3. ОС LINUX: ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ КОМАНДНОГО ИНТЕРПРЕТАТОРА. ОДНОНАПРАВЛЕННАЯ ПЕРЕДАЧА ТЕКСТОВОЙ ИНФОРМАЦИИ

## 3.1. Цель лабораторной работы

Целью данной работы является изучение средств организации взаимодействия процессов из состава средств System V IPC.

## 3.2. Перечень компетенций, формируемых в ходе выполнения лабораторной работы

Профессиональная компетенция ПК-1:

ПК-1.2.3

Умеет использовать командные языки для отладки и запуска программ системной обработки в соответствии с заданием (однонаправленная передача текстовой информации)

## 3.3. Задание на выполнение лабораторной работы

1. Разработать две программы 1 a.c и 1b.c:

а) Первая посылает пять текстовых сообщений с типом 1 и одно сообщение нулевой длины с типом 255 второй программе.

б) Вторая программа в цикле принимает сообщения любого типа в порядке FIFO и печатает их содержимое до тех пор, пока не получит сообщение с типом 255. Сообщение с типом 255 служит для нее сигналом к завершению работы и ликвидации очереди сообщений. Если перед запуском любой из программ очередь сообщений еще отсутствовала в системе, то программа создаст ее.

2) Выполните компиляцию файлов 1a.c и 1b.c и проверьте правильность их поведения.

**Обратите внимание** на использование сообщения с типом 255 в качестве сигнала прекращения работы второго процесса. Это сообщение имеет нулевую длину, так как его информативность исчерпывается самим фактом наличия сообщения.

3) Выполните модификацию предыдущих программ для передачи числовой информации.

## 3.4. Основные теоретические сведения

### 3.4.1. Сообщения как средства связи и средства синхронизации процессов

Существуют такие средства организации взаимодействия процессов из состава средств System V IPC, как разделяемая память и семафоры. Кроме этого, существуют очереди сообщений, как наиболее семантически нагруженные средства, входящие в System V IPC.

Можно рассматривать модели сообщений как способ взаимодействия процессов через линии связи, в котором на передаваемую информацию накладывается определенная структура, так что процесс, принимающий данные, может четко определить, где заканчивается одна порция информации и начинается другая. Такая модель позволяет задействовать одну и ту же линию связи для передачи данных в двух направлениях между несколькими процессами. Существует возможность использования сообщений с встроенными механизмами взаимоисключения и блокировки при чтении из пустого буфера и записи в переполненный буфер для организации синхронизации процессов.

### 3.4.2. Очереди сообщений в UNIX как составная часть System V IPC

Очереди сообщений, как и семафоры, и разделяемая память, являются средством связи с непрямой адресацией, требуют инициализации для организации взаимодействия процессов и специальных действий для освобождения системных ресурсов по окончании взаимодействия.

Пространством имен очередей сообщений является то же самое множество значений ключа, генерируемых с помощью функции ftok(). Очереди сообщений располагаются в адресном пространстве ядра операционной системы в виде однонаправленных списков и имеют ограничение по объему информации, хранящейся в каждой очереди. Каждый элемент списка представляет собой отдельное сообщение. Сообщения имеют атрибут, называемый *типом сообщения*. Выборка сообщений из очереди (выполнение примитива receive ) может осуществляться тремя способами:

- в порядке FIFO, независимо от типа сообщения;
- в порядке FIFO для сообщений конкретного типа;
- в первым выбирается сообщение с минимальным типом, не превышающим некоторого заданного значения, пришедшее раньше других сообщений с тем же типом.

Очереди сообщений, как и другие средства System V IPC, позволяют организовать взаимодействие процессов, не находящихся одновременно в вычислительной системе.

### 3.4.3. Создание очереди сообщений или доступ к уже существующей. Системный вызов msgget()

Для создания очереди сообщений, ассоциированной с определенным ключом, или доступа по ключу к уже существующей очереди используется системный вызов msgget() , являющийся аналогом системных вызовов shmget() для разделяемой памяти и semget() для массива семафоров, который возвращает значение IPC-дескриптора для этой очереди. При этом существуют те же способы создания и доступа, что и для разделяемой памяти или семафоров.

**а) Системный вызов msgget()**

Прототип системного вызова:

*#include <types.h>*

*#include <ipc.h>*
*#include <msg.h>*
*int msgget(key_t key, int msgflg);*

Системный вызов msgget предназначен для выполнения операции доступа к очереди сообщений и, в случае ее успешного завершения, возвращает дескриптор System V IPC для этой очереди (целое неотрицательное число, однозначно характеризующее очередь сообщений внутри вычислительной системы и использующееся в дальнейшем для других операций с ней).

Параметр key является ключом System V IPC для очереди сообщений, т. е. фактически ее именем из пространства имен System V IPC. В качестве значения этого параметра может быть использовано значение ключа, полученное с помощью функции ftok(), или специальное значение IPC_PRIVATE. Использование значения IPC_PRIVATE всегда приводит к попытке создания новой очереди сообщений с ключом, который не совпадает со значением ключа ни одной из уже существующих очередей и не может быть получен с помощью функции ftok() ни при одной комбинации ее параметров.

Параметр msgflg – флаги - играет роль только при создании новой очереди сообщений и определяет права различных пользователей при доступе к очереди, а также необходимость создания новой очереди и поведение системного вызова при попытке создания. Он является некоторой комбинацией следующих предопределенных значений и восьмеричных прав доступа:

- IPC_CREAT - если очереди для указанного ключа не существует, она должна быть создана;

- IPC_EXCL - применяется совместно с флагом IPC_CREAT. При совместном их использовании и существовании массива с указанным ключом доступ к очереди не производится и констатируется ошибочная ситуация, при этом переменная errno, описанная в файле <errno.h>, примет значение EEXIST:

0400 - разрешено чтение для пользователя, создавшего очередь;

0200 - разрешена запись для пользователя, создавшего очередь;

0040 - разрешено чтение для группы пользователя, создавшего очередь;

0020 - разрешена запись для группы пользователя, создавшего очередь;

0004 - разрешено чтение для всех остальных пользователей;

0002 - разрешена запись для всех остальных пользователей;

Очередь сообщений имеет ограничение по общему количеству хранимой информации, которое может быть изменено администратором системы. Текущее значение ограничения можно узнать с помощью команды *ipcs –l.*

Системный вызов возвращает значение дескриптора System V IPC для очереди сообщений при нормальном завершении и значение -1 при возникновении ошибки.

**б) Системные вызовы msgsnd() и msgrcv()**

Для выполнения примитива send используется системный вызов **msgsnd()** , копирующий пользовательское сообщение в очередь сообщений, заданную

IPC-дескриптором. При изучении описания этого вызова обратите особое внимание на следующие моменты:

Тип данных struct msgbuf не является типом данных для пользовательских сообщений, а представляет собой лишь шаблон для создания таких типов. Пользователь создает сам структуру для своих сообщений, в которой первым полем должна быть переменная типа long, содержащая положительное значение типа сообщения.

В качестве третьего параметра (длина сообщения) указывается не вся длина структуры данных, соответствующей сообщению, а только длина полезной информации, т. е. информации, располагающейся в структуре данных после типа сообщения. Это значение может быть и равным 0 в случае, когда вся полезная информация заключается в самом факте прихода сообщения (сообщение используется как сигнальное средство связи).

В учебных целях,как правило, будем использовать нулевое значение флага системного вызова, которое приводит к блокировке процесса при отсутствии свободного места в очереди сообщений.

**в) Системный вызов msgsnd()**

Прототип системного вызова:

*#include <sys/types.h>*
*#include <sys/ipc.h>*
*#include <sys/msg.h>*
*int msgsnd(int msqid, struct msgbuf *ptr,*
*int length, int flag);*

Системный вызов msgsnd предназначен для помещения сообщения в очередь сообщений, т. е. является реализацией примитива send.

Параметр msqid является дескриптором System V IPC для очереди, в которую отправляется сообщение, т. е. значением, которое вернул системный вызов msgget() при создании очереди или при ее поиске по ключу.

Структура struct msgbuf описана в файле <sys/msg.h> в виде:

*struct msgbuf {*
*long mtype;*
*char mtext[1];*
*};*

Она представляет собой некоторый шаблон структуры сообщения пользователя. Сообщение пользователя –- это структура, первый элемент которой обязательно имеет тип long и содержит тип сообщения, далее следует информативная часть теоретически произвольной длины (практически в Linux она ограничена размером 4080 байт и может быть еще уменьшена системным администратором), содержащая собственно суть сообщения.

Например:

*struct mymsgbuf {*
*long mtype;*
*char mtext[1024];*

*} mybuf;*

При этом информация не обязана быть текстовой:

*struct mymsgbuf {*
*long mtype;*

### 3.4.4. Удаление очереди сообщений из системы с помощью команды ipcrm или системного вызова msgctl()

После завершения процессов, использовавших очередь сообщений, она не удаляется из системы автоматически, а продолжает сохраняться в системе вместе со всеми невостребованными сообщениями до тех пор, пока не будет выполнена специальная команда или специальный системный вызов. Для удаления очереди сообщений можно воспользоваться командой ipcrm, которая в этом случае примет вид:

*ipcrm msg <IPC идентификатор>*

Для получения IPC идентификатора очереди сообщений необходимо использовать команду ipcs.

Можно удалить очередь сообщений и с помощью системного вызова msgctl() . Этот вызов умеет выполнять и другие операции над очередью сообщений. Если какой-либо процесс находился в состоянии «ожидание» при выполнении системного вызова msgrcv() или msgsnd() для удаляемой очереди, то он будет разблокирован, и системный вызов констатирует наличие ошибочной ситуации.

### 3.4.5. Системный вызов msgctl()

Прототип системного вызова:

*#include <sys/types.h>*
*#include <sys/ipc.h>*
*#include <sys/msg.h>*
*int msgctl(int msqid, int cmd,*
*struct msqid_ds *buf);*

Системный вызов msgctl предназначен для получения информации об очереди сообщений, изменения ее атрибутов и удаления из системы. Данное описание не является полным описанием системного вызова, для изучения полного описания можно воспользоваться UNIX Manual.

В данной лабораторной работе системный вызов msgctl будет использоваться только для удаления очереди сообщений из системы. Параметр msqid является дескриптором System V IPC для очереди сообщений, т. е. значением, которое вернул системный вызов msgget() при создании очереди или при ее поиске по ключу.

В качестве параметра cmd будем передавать значение IPC_RMID - команду для удаления очереди сообщений с заданным идентификатором. Параметр buf для этой команды не используется, будем подставлять туда значение NULL.

Системный вызов возвращает значение 0 при нормальном завершении и значение -1 при возникновении ошибки.

### 3.4.6. Примеры программ однонаправленной передача текстовой информации

Текст программы1a.c для иллюстрации работы с очередями сообщений приведена в примере 7. Эта программа получает доступ к очереди сообщений, отправляет в нее 5 текстовых сообщений с типом 1 и одно пустое сообщение с типом 255, которое будет служить для программы 1b.c сигналом прекращения работы.

Пример 7.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>
#include <stdio.h>
/* Тип сообщения для прекращения работы программы 1b.c */
#define LAST_MESSAGE 255
int main()
{
/* IPC дескриптор для очереди сообщений */
   int msqid;
/* Имя файла, использующееся для генерации ключа. Файл с таким
      именем должен существовать в текущей директории */
  char pathname[] = "1a.c";
/* IPC ключ */
   key_t key;
/* Счетчик цикла и длина информативной части сообщения */
   int i,len;
   /* Ниже следует пользовательская структура для сообщения */
   struct mymsgbuf
   {
      long mtype;
      char mtext[81];
   } mybuf;
   /* Генерируем IPC ключ из имени файла 09-1a.c в текущей директории и
номера экземпляра очереди сообщений 0 */
   if((key = ftok(pathname,0)) < 0){
      printf("Can\'t generate key\n");
      exit(-1);
   }
   /* Попытка получить доступ по ключу к очереди сообщений, если она
существует, или создать ее, с правами доступа read & write для всех
пользователей */
   if((msqid = msgget(key, 0666 | IPC_CREAT)) < 0){
      printf("Can\'t get msqid\n");
```

```
    exit(-1);
  }
  /* Посылаем в цикле 5 сообщений с типом 1   в очередь сообщений,
идентифицируемую msqid.*/
  for (i = 1; i <= 5; i++){
    /* Сначала заполняем структуру для создаваемого  сообщения и
определяем длину информативной части */
    mybuf.mtype = 1;
    strcpy(mybuf.mtext, "This is text message");
    len = strlen(mybuf.mtext)+1;
    /* Отсылаем сообщение. В случае ошибки сообщаем об  этом и удаляем
очередь сообщений из системы. */
    if (msgsnd(msqid, (struct msgbuf *) &mybuf,
      len, 0) < 0){
      printf("Can\'t send message to queue\n");
      msgctl(msqid, IPC_RMID,
        (struct msqid_ds *) NULL);
      exit(-1);
    }
  }
  /* Отсылаем сообщение, которое заставит получающий процесс
прекратить работу, с типом LAST_MESSAGE и длиной 0 */
  mybuf.mtype = LAST_MESSAGE;
  len = 0;
  if (msgsnd(msqid, (struct msgbuf *) &mybuf,
    len, 0) < 0){
    printf("Can\'t send message to queue\n");
    msgctl(msqid, IPC_RMID,
      (struct msqid_ds *) NULL);
    exit(-1);
  }
  return 0;
```

Текст программы1b.c для иллюстрации работы с очередями сообщений приведен в примере 8. Эта программа получает доступ к очереди сообщений и читает из нее сообщения с любым типом в порядке FIFO до тех пор, пока не получит сообщение с типом 255, которое будет служить сигналом прекращения работы.

Пример 8.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>
```

```
#include <stdio.h>
/* Тип сообщения для  прекращения работы */
#define LAST_MESSAGE 255
int main()
{
/* IPC дескриптор для очереди сообщений */
   int msqid;
/* Имя файла, использующееся для генерации ключа. Файл с таким именем
должен существовать в текущей директории */
   char pathname[] = "1a.c";
/* IPC ключ */
   key_t key;
/* Реальная длина и максимальная  длина информативной части сообщения */
   int len, maxlen;
   /* Ниже следует пользовательская структура для сообщения */
   struct mymsgbuf
   {
      long mtype;
      char mtext[81];
   } mybuf;
   /* Генерируем IPC ключ из имени файла 1a.c в текущей  директории и
номера экземпляра очереди сообщений 0 */
   if((key = ftok(pathname,0)) < 0){
      printf("Can\'t generate key\n");
      exit(-1);
   }
   /* Попытка получить доступ по ключу к очереди сообщений, если она
существует, или создать ее, с правами доступа
   read & write для всех пользователей */
   if((msqid = msgget(key, 0666 | IPC_CREAT)) < 0){
      printf("Can\'t get msqid\n");
      exit(-1);
   }
   while(1){
      /* В бесконечном цикле принимаем сообщения  любого типа в порядке
FIFO с максимальной длиной информативной  части 81 символ до тех пор,
пока не поступит сообщение с  типом LAST_MESSAGE*/
      maxlen = 81;
      if(( len = msgrcv(msqid,
         (struct msgbuf *) &mybuf, maxlen, 0, 0) < 0){
         printf("Can\'t receive message from queue\n");
         exit(-1);
      }
```

```
    /* Если принятое сообщение имеет тип LAST_MESSAGE,  прекращаем
работу и удаляем очередь сообщений из  системы. В противном случае
печатается текст принятого  сообщения. */
      if (mybuf.mtype == LAST_MESSAGE){
         msgctl(msqid, IPC_RMID,
         (struct msqid_ds *) NULL);
      exit(0);
      }
      printf("message type = %ld, info = %s\n",
      mybuf.mtype, mybuf.mtext);
   }
/* Для отсутствия warning'ов при компиляции. */
   return 0;
}
```

### 3.4.7. Модификация предыдущего примера для передачи числовой информации

Передаваемая информации не обязательно должна представлять собой текст. Можно воспользоваться очередями сообщений для передачи данных любого вида. При передаче разнородной информации целесообразно информативную часть объединять внутри сообщения в отдельную структуру:

```
struct mymsgbuf {
long mtype;
struct {
 short sinfo;
 float finfo;
} info;
} mybuf;
```

для правильного вычисления длины информативной части. В некоторых вычислительных системах числовые данные размещаются в памяти с выравниванием на определенные адреса (например, на адреса, кратные 4). Поэтому реальный размер памяти, необходимой для размещения нескольких числовых данных, может оказаться больше суммы длин этих данных, т. е.

$sizeof(info) >= sizeof(short) + sizeof(float)$

Для полной передачи информативной части сообщения в качестве длины нужно указывать не сумму длин полей, а полную длину структуры, например при реализации двусторонней связи через одну очередь сообщений.

Наличие у сообщений типов позволяет организовать двустороннюю связь между процессами через одну и ту же очередь сообщений. Процесс 1 может посылать процессу 2 сообщения с типом, а получать от него сообщения с типом 2. При этом для выборки сообщений в обоих процессах следует пользоваться вторым способом выбора.

### 3.5. Вопросы к защите лабораторной работы

1) Какой объект позволяет организовывать взаимодействие процессов, не находящихся одновременно в вычислительной системе?
2) Опишите метод FIFO.
3) Дайте определение очереди сообщений.
4) Что такое IPC?
5) Опишите структуру ядра msg.
6) Назначение команды IPC_RMID в программе.
7) Что представляет собой структура msg_buf?
8) Назначение системных вызовов.
9) Что возвращает msgget в случае успешного вызова?
10) Чем определяется емкость очереди и где она инициализируется?

### 3.6. Список рекомендуемой литературы

1) Робачевский А. Операционная система UNIX.-СПб.:BHV-Санкт-Петербург, 2010- 528 с.
2) Романчева Н.И. Системное программное обеспечение. Учебное пособие. – М.: МГТУ ГА, 2013.
3) Материалы сайта ИНТУИТ / Эл. ресурс URL: https://www.intuit.ru (дата размещения - постоянно).