МИНИСТЕРСТВО ТРАНСПОРТА РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное образовательное учреждение высшего профессионального образования
МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ГРАЖДАНСКОЙ АВИАЦИИ

Кафедра вычислительных машин, комплексов, систем и сетей

Н.И. РОМАНЧЕВА

оглавление

ПОСОБИЕ

к выполнению лабораторных работ № 3, 4

по дисциплине

«СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ»

для студентов 3 курса

специальности 220100

дневного обучения

Москва- 2004

Рецензент канд..техн.наук М.М. Константиновский, ген. директор ООО ИТК “Феникс”

Романчева Н.И., канд. техн.наук, доцент

Пособие к выполнению лабораторных работ № 3,4 по дисциплине "Системное программное обеспечение". - М.: МГТУ ГА, 2004.- 48 с.

Данное методическое пособие издается в соответствии с учебным планом для студентов специальности 220100 дневного обучения.

Рассмотрено и одобрено на заседаниях кафедры 09.03.2004г. и Методического совета по специальности 220100 09.03.2004 г.

С О Д Е Р Ж А Н И Е

1 ОСНОВНЫЕ ТРЕБОВАНИЯ И ПОРЯДОК ВЫПОЛНЕНИЯ ЛАБОРАТОРНЫХ РАБОТ

Настоящее пособие предназначено для студентов специальности 220100, выполняющих лабораторные работы по дисциплине "Системное программное обеспечение". В пособие включены материалы по лабораторным работам № 3, 4.

Продолжительность каждой лабораторной работы - 4 часа.

Целью проведения лабораторных работ является закрепление основных теоретических положений, изложенных в лекциях на примере широко используемых в различных областях ОС UNIX (ASPLinux).

В процессе выполнения лабораторных работ студенты должны получить практические навыки программирования на языке командного интерпретатора, а также навыки отладки программ на языке С для 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;

5) защита лабораторной работы.

В процессе домашней подготовки студент:

- изучает лекционный материал, материалы по темам данного пособия и дополнительной литературы,
- знакомится с заданием на выполнение лабораторной работы;
- готовит отчет по выполнению лабораторной работы (пункты, отмеченные знаком *).

Выполнение лабораторной работы производится во время занятий в классе ЛВС кафедры ВМКСС МГТУГА в присутствии преподавателя. 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 задания.

После приема преподавателем лабораторной работы на ПК студент:

- сохраняет результаты лабораторной работы на дискете, выданной преподавателем, в каталоге со своей фамилией;
- распечатывает результаты на принтере на подготовленных листах формата А4.

Отчет по каждой лабораторной работе должен содержать:

- название работы*;
- цель лабораторной работы*;
- задание на выполнение лабораторной работы*;
- алгоритмы программ,
- распечатки файлов результатов, подписанные преподавателем.

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

2 ЛАБОРАТОРНАЯ РАБОТА №3 ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ КОМАНДНОГО ИНТЕРПРЕТАТОРА

2.1 Цель работы

Целью данной работы является изучение основ программирования и отладки сценариев на языке командного интерпретатора Bourne Shell (использование условных выражений и логических операторов and/or в циклах while и until, программное прерывание бесконечного цикла, установка кода завершения, использование функций в сценариях).

2.2 Задание на выполнение работы

1) Просмотреть таблицы дескрипторов процесса.

2) Запустить программу поиска заданного слова в файле паролей.

3) Используя условные выражения, написать и отладить скрипт, проверяющий наличие в домашнем каталоге инициализационного скрипта .profile, а в случае его отсутствия выполнить копирование шаблона.

4) Написать скрипт очистки неиспользуемых файлов (за определенный промежуток времени) во временных каталогах (/tmp, /usr/tmp, /home/student/tmp).

5) Написать и отладить скрипт, иллюстрирующий различные способы защиты файлов. Для контроля (чтения) использовать команду cat:

- установить для файла только право на запись;
- для каталога установлено только право на чтение;
- вывести список и прочитать файл (должно быть - файлы видно, нельзя прочитать содержимое);
- установить для каталога бит использования (выполнения, execute bit);

- прочитать содержимое файла и удалить (должно быть - файлы читаются, но не удаляются);
- установить корректно права на доступ к каталогу и файл удалить.

6) Используя операторы цикла while, until, написать скрипт, выводящий на экран целые числа в заданном диапазоне.

7) Отладить пример кода с использованием логических операторов AND и OR.

8) Используя команды for и bc вывести квадратные корни чисел в заданном диапазоне.

9) Написать и отладить программу, реализующую алгоритм угадывания числа.

10) Написать и отладить программу, демонстрирующую перехват прерываний при выходе.

11) Разработать функцию, удаляющую временные файлы при завершении работы сценария.

2.3 Краткие теоретические сведения

В интерпретатор встроен мощный язык программирования, используемый для самых разных задач: от автоматизации повторяющихся команд до написания сложных интерактивных программ обработки данных. По соглашению в именах локальных переменных используются символы нижнего регистра, а в именах переменных среды – верхнего.

2.3.1 Условные операторы

Условные операторы выполняются в том случае, когда определенное условие (или несколько условий) истинно. Они могут принимать одну из трех общих форм: if, case и AND/OR.

Операторы if

Операторы if проверяют числовые выражения. Если условие истинно,

выполняются операторы внутри блока if. Если оно ложно, то возможно два варианта действий:

1) операторы внутри блока не запускаются, и программа продолжает выполняться дальше;

2) если в блок if включен оператор else, выражения из последнего выполняются, если условие ложно, т.е., поток управления программы следует правилу: "Сделать это, если условие истинно, или то, если оно ложно, но не оба действия одновременно".

Синтаксис оператора:

```
if  условие
 then
        команда1
        команда2
fi
```

Более сложный формат:

```
if  условие
 then
        команда1
elif   условие
        команда 2
else
        команда 3
fi
```

Например, в следующей программе оператор if используется для проверки количества аргументов командной строки, переданной программе. Если их число больше или равно 1, программа выполняет операции внутри блока. Если они отсутствуют, программа завершает работу, не выполняя никаких действий.

```
1.  #!/bin/sh
2.  # Пример использования оператора if
```

```
3.    if  [  $# -ge  1  ]
4.    then
5.          echo  “You  supplied  $# command  line arguments.”
6.    fi
7.    echo
8.    echo   “Program  exiting…”
9.    echo
10.   exit  0
```

Пример запуска программы:

```
./test   file1  file2   file3
```

где *test* – имя файла (скрипта), *file1 file2 file3* – аргументы.

Оператор if в строке 3 проверяет число аргументов командной строки, используя переменную $#, в которой оно содержится. Если оно больше или равно 1, выполняются операторы, заключенные между ключевыми словами then и fi (fi - это if наоборот, данный оператор отмечает конец блока). Если нет, управление передается оператору, следующему за fi. В данном случае он информирует пользователя о выходе из программы.

Часть then оператора if является обязательной, а часть else — нет. В рассмотренном примере операторы в блоке then выполняются тогда, когда условие истинно. Но иногда возникает необходимость выполнить действия лишь тогда, когда выражение ложно. В этом случае нужно воспользоваться двоеточием. Например:

```
if   [ $myvar -gt 5 ]
then
        : # Не предпринимать никаких действий, выйти из блока if
else
        # Операторы,  которые выполняются,  если условие ложно.
fi
```

В некоторых случаях возникает необходимость проверить два или несколько разных условий и предпринять различные действия в зависимости от результатов каждого этапа. Для этих целей используется оператор elif (аббревиатура от else if). Когда используется оператор elif, программа вначале выполняет оператор if. Если его условие истинно, выполняется его код, а затем управление передается следующему оператору (т.е. оператору, расположенному после fi). Если условие ложно, проверяется условие в первом операторе elif. Если оно истинно, выполняются операторы из его блока, и программа переходит к концу блока if. Если оно ложно, проверяется следующий оператор elif и т.д. Фактически, условия проверяются до тех пор, пока одно из них не даст значение истина. Если такого условия нет, ничего не происходит, либо запускаются операторы, заключенные в блоке else (если он присутствует).

Операторы if поддерживают также логические операторы AND (&&) и OR (||).

2.3.2 Логические операторы AND/OR

Логические условные операторы AND/OR в некоторых случаях заменяют операторы if. Код завершения первой команды используется как условие запуска второй. Например:

tar cvfz backup.tar.gz documents/2004/ && rm -r documents/2004*

Эта команда означает следующее: "Если первая операция прошла успешно, выполнить вторую. Если нет — вторая команда не выполняется". Другими словами: "Необходимо выполнить команды A и B. Но если команда A невыполнима, то не следует исполнять и B". В данном случае первая команда архивирует все файлы из каталога documents/2004 в файл backup.tar.gz. Если этот процесс завершается успешно (команда tar возвращает код 0), выполняется команда после оператора &&, которая удаляет каталог documents/2004. Если же процесс архивирования не завершен успешно (команда tar возвращает код, отличный от 0), команда

после && не исполняется (нет смысла удалять каталог, если он не был корректно заархивирован).

Символом || обозначается оператор OR. Он означает следующее: "Если A невыполнимо, исполнить B. Но если A завершилось успешно, B не выполнять". Например:

tar cvfz backup.tar.gz doc/2004/ || echo "Archive operation failed."*

В этом случае, если операция архивирования завершается успешно (tar возвращает 0), команда после оператора || не запускается. Если же нет (tar возвращает значение, отличное от 0), выполняется команда, указанная после ||, - на экран выводится сообщение об ошибке.

2.3.3 Организация циклов

В некоторых случаях требуется повторять действие до тех пор, пока определенное выражение не станет истинным (или, наоборот, перестанет быть истинным). Язык программирования Bourne shell поддерживает три вида циклических конструкций.

1) Цикл while

Синтаксис:

while условие
do
команда1
команда2
….
done

Цикл while выполняет операторы, заключенные в нем, до тех пор, пока условие цикла является истинным. Если уже при первом проходе цикла условие ложно, операторы внутри цикла выполняться не будут. Например, в следующей программе цикл while используется для вывода на экран целых чисел от 1 до 20.

1. #!/bin/sh

```
2. # Count  from  1 to 20
3. i=1
4. while  [  $I  -le  20 ]
5. do
6.          echo  $I
7.         i=’expr  $I  + 1’
8. done
9. exit  0
```

В этом примере в строке 3 переменной i присваивается начальное значение 1. i часто применяется как счетчик цикла, поэтому здесь нет необходимости использовать описательное имя.

Строка 4: Команда while содержит условие, заключенное в квадратные скобки. На самом деле, они представляют собой сокращенную запись команды test, которая часто используется в сценариях командного интерпретатора. Команда test использует достаточно прозрачный синтаксис, -1e в данном примере обозначает "меньше или равно". Таким образом, цикл выполняется до тех пор, пока значение переменной i меньше или равно 20. Операторы сравнения, поддерживаемые данной командой, приведены в таблице 1.

Таблица 1 – Операции сравнения команды test

Опции	Значение
-eq	Истина, если операнды равны
-ne	Истина, если операнды не равны
-gt	Истина, если первый операнд больше второго
-ge	Истина, если первый операнд больше или равен второму
-lt	Истина, если первый операнд меньше второго
-le	Истина, если первый операнд меньше или равен второму

Строка 5: Оператор do указывает, что все операторы, следующие за ним, должны выполняться при каждой итерации цикла. Все выражения между do и done являются телом цикла.

Строка 6: Здесь выводится значение переменной i.

Строка 7: Здесь используется подстановка команд, значение переменной i увеличивается на единицу, а затем новое значение вновь присваивается

переменной.

Строка 8: Завершающий оператор цикла. В этой точке программа вновь возвращается к оператору while и вновь проверяет условие цикла. Если значение переменной i все еще меньше или равно 20, операторы в теле цикла выполняются снова. Если i больше 20, цикл завершается и управление передается оператору, следующему за оператором done (в данном случае, это просто оператор exit в строке 9, который завершает программу с кодом успешного выполнения, равным 0).

Операторы внутри цикла выровнены с отступом по левому краю. Это позволяет легко выделить цикл при чтении исходного кода. Интерпретатор игнорирует отступ, выполняя команды обычным образом. Пробел между [и тестируемым условием является обязательным. Его отсутствие приведет к ошибке. Например, [$VarA -gt 5] выполняется, а [$VarA -gt 5] возвращает ошибку.

2) <u>Цикл until</u>

Синтаксис:

until условие
do
    команда1
    команда2
    . . . .
done

Цикл until по смыслу противоположен циклу while. Он выполняет последовательность операций до тех пор, пока условие не станет истинным. В этом случае цикл завершается. Если условие истинно уже при первом запуске, операторы в теле цикле не запускаются. Циклы while и until очень похожи. Как правило, любой из них можно использовать в программе и добиваться одного и того же результата, изменяя условие. Например, предыдущую программу можно переписать, заменив цикл while на until. При этом в строке 4 потребуется лишь два изменения:

```
until  [  $I   -gt    20  ]
```

Программа выполняет ту же функцию. Единственное различие заключается в том, что теперь цикл выполняется до тех пор, пока значение переменной i не станет большим 20, тогда как цикл while выполнялся до тех пор, пока i было меньше или равно 20. В обоих случаях программа дает один и тот же результат.

Логические операторы AND/OR в циклах while и until

Циклы while и until позволяют работать с логическими операторами AND/OR. Логическое выражение AND возвращает значение истина лишь в том случае, когда оба операнда истинны, а выражение OR - когда лишь один из операндов имеет значение истина. Ниже приведен пример кода с логическим оператором AND:

```
VarA=1
VarB=5
while [ $VarA -eg 1 ]  &&  [ $VarB -gt 7 ]
do
        echo "VarA is equal to 1 and VarB is greater than 7"
done
```

В этом примере оператор echo не запускается, так как $VarA равно 1, а $VarB не больше 7. Поскольку цикл while в данном случае требует, чтобы оба условия были истинными, тест не проходит и возвращает значение 0 (ложь).

3) Цикл for

Синтаксис:

```
for  var  in  список
do
        команда1
        команда2
        ……..
done
```

Оператор for обеспечивает выполнение цикла столько раз, сколько слов в *списке*. При этом переменная var последовательно принимает значения, равные словам из списка. Список может формироваться различными способами, например, как вывод некоторой команды (‘имя_ команды_формирующей _список’) или с помощью шаблонов shell.

В другой форме for, когда список отсутствует, переменная var принимает значения позиционных параметров, переданных скрипту.

Shift

Команда shift похожа на цикл for. Чтобы выполнить цикл один раз для каждого аргумента командной строки, переданного сценарию, можно воспользоваться оператором while и командой shift.

Аргументы командной строки хранятся в переменных от $1 до $9. Каждый запуск команды shift сдвигает переменные на одну позицию влево. Это значит, например, что информация, сохраненная в $1, отбрасывается, а значение переменной $2 присваивается $1. Например:

```
1.  #!/bin/sh
2.  # Программа иллюстрирует применение команды shift.
3.  while [ $# -ne 0 ]
4.  do
5.      echo "The value of \$1 is now $1."
6.      shift
7.  done
8.  echo
9.  exit 0
```

Ниже приведен пример запуска и результат программы.

```
bash$ ./file a b c d
The value of $1 is now a
The value of $1 is now b
The value of $1 is now c
The value of $1 is now d
```

Строка 3: Здесь начинается цикл while. Переменная $# содержит общее число аргументов командной строки. Цикл while выполняется до тех пор, пока значение $# не становится равным нулю. Если $# равно нулю, следовательно, все аргументы использованы, после чего цикл завершается.

Строка 5: В этой строке выводится текущее значение переменной $1. Обратите внимание, что для печати строкового значения $1 на экране символ $ необходимо экранировать символом обратной косой черты, поскольку сам по себе он имеет специальное значение.

Строка 6: После выполнения команды shift переменные сдвигаются на одну позицию влево. Значение $1 отбрасывается (оно больше недоступно), $2 смещается в $1, $3 - в $2 и т.д.

Одним из распространенных вариантов применения команд shift (и циклов for) является обработка имен файлов, заданных как аргументы командной строки, и выполнение операций над каждым из них.

2.3.4 Перехват прерываний при выходе

Как известно, выполнение программы можно прервать, послав какой-либо из сигналов kill, а также посредством различных комбинаций клавиш (например, Ctrl-C). Проблема заключается в том, что если программа создает временные файлы, а пользователь прерывает ее работу по Ctrl-C, созданные файлы не будут удалены. За короткий срок таких файлов может накопиться достаточно много. Командный интерпретатор позволяет перехватывать подобные прерывания. Ниже приведен короткий пример, иллюстрирующий перехват прерываний:

```
#!/bin/sh
# Программа демонстрирует перехват прерываний
trap 'echo "Interrupt received. Quitting." 1>&2' 1 2 3 15
echo -n "Enter a number: "
readln num
exit 0
```

Программа устанавливает перехват прерываний 1, 2, 3 и 15. Предпринимаемые действия задаются в одинарных кавычках. Если при запуске программы, после того как она выдаст приглашение "Enter a number", нажать Ctrl-C, она получит сигнал 2 (INT). Поскольку программа перехватывает это прерывание, она выдаст сообщение "Interrupt received" и завершит работу.

В приведенном примере следует обратить внимание на использование команды echo. Фактически, это перенаправление вывода, обеспечиваемое самим интерпретатором. Команда 1>&2 в операторе echo перенаправляет вывод в поток STDERR. Поэтому специальное сообщение нельзя случайно перенаправить другой команде с помощью конвейера или в файл вместе с остальным выводом программы. Рекомендуется все сообщения об ошибках перенаправлять в поток STDERR командой 1>&2.

Обычно перехват прерываний используется для операций типа удаления временных файлов. Если требуется выполнить несколько действий, желательно воспользоваться функцией, перехватывающей прерывание. Если вы хотите исключить возможность выхода из программы по Ctrl-C, можно установить перехват прерывания, не исполняющий никаких действий. Например:

trap '' 2

Данный оператор приводит к тому, что сигнал 2 полностью игнорируется.

Программа может перехватывать несколько прерываний и выполнять различные действия в зависимости от того, как осуществляется выход из нее. Перехват сигнала 0 установлен для всех вариантов выхода из программы. Перехват остальных сигналов осуществляется лишь тогда, когда они посылаются программе. Список наиболее часто перехватываемых сигналов приведен ниже:

0 - выход; 1 – HUP – обрыв сеанса (или отсоединение);

2 – INT – прерывание (Ctrl-C); 3 – Quit - выход (Ctrl -\)$

15 – TERM –обычная команда kill.

Сигнал 15 (посылаемый по умолчанию командой kill) и другие сигналы команды kill можно перехватывать, однако это не относится к сигналу 9 (SIGKILL). Он используется в качестве последней возможности прервать работу программы, когда остальные методы не помогают. Поэтому его нельзя ни перехватить, ни игнорировать.

2.3.5 Функции

Функции представляют собой группы операторов, вызываемые одной командой. Их можно рассматривать как "мини-программы внутри программ". Использование функций в сценариях облегчает программирование по двум причинам. Прежде всего, если определенный набор операций требуется выполнить в нескольких местах программы, достаточно воспользоваться лишь одной командой - именем функции. Во-вторых, если вы захотите изменить то, как исполняется операция, достаточно будет шести изменения лишь в одном фрагменте кода — в теле функции.

Для создания функции необходимо задать ее имя, за которым следуют круглые скобки и открывающая фигурная скобка. Все, что находится между фигурными скобками, представляет собой тело функции. Функция вызывается точно так, как и любая команда, - по имени. При этом выполняются все операторы, заключенные в фигурные скобки.

Ниже приведен пример функции, которая удаляет временные файлы при завершении работы сценария.

```
#!/bin/sh
on_exit () {
        rm -rf /tmp/myprogram.*
        mv logfile logfile.old
        mail foo@bar.com < report.txt
}
```

Пример вызова данной функции:

trap on_exit 0 1 2 3 15

Между вызовом функции и вызовом другой программы существует важное различие. Функция выполняется текущим интерпретатором, а отдельная программа запускается в другой копии командного интерпретатора. Это значит, что функциям доступны и переменные среды, и внутренние переменные вызывающей ее программы. Отдельной программе, исполняемой другим интерпретатором, они недоступны.

2.3.6 Файловые дескрипторы

Файловые дескрипторы представляют собой числовые идентификаторы, устанавливаемые ядром при запуске каждого нового процесса. Процесс использует их для записи вывода и чтения ввода. По умолчанию командный интерпретатор открывает три файловых дескриптора:

- 0 - это стандартный входной поток STDIN. Обычно ввод поступает с клавиатуры, однако его можно перенаправить из файла или какого-либо другого источника;
- 1 - это стандартный выходной поток STDOUT. Обычно вывод поступает на экран, однако, его также можно перенаправить;
- 2 – это стандартный поток ошибок STDERR. Обычно выводится на экран, но и его можно перенаправить.

Чаще всего числовое значение дескриптора, являющееся указателем на соответствующий поток, используется для потока ошибок. Например, чтобы подавить вывод ошибок, можно использовать следующую запись:

$ run 2>/dev/null

где /dev/null – является псевдоустройством, удаляющим все введенные в него символы.

Командный интерпретатор предоставляет возможность слияния потоков. Например, при запуске команды

$ run testprog > /dev/null 2>&1 &

сообщения об ошибках будут также выводиться в файл /dev/null. Символ & перед именем потока необходим, чтобы отличить его от файла с именем 1. В данном примере изменение порядка двух перенаправлений потоков приведет к тому, что сообщения об ошибках по-прежнему будут выводиться на экран: Shell анализирует командную строку слева направо, таким образом сначала будет осуществлено слияние потоков и оба будут указывать на терминал пользователя, а затем стандартный поток вывода (1) будет перенаправлен в файл /dev/null. Использование символа & в конце команды переводит задание в фоновый режим.

2.3.7 Отладка сценариев командного интерпретатора

Интерпретатор, не имея полноценного отладчика, обеспечивает простейшие возможности для мониторинга всех выполняемых действий. Трассировка включается посредством опции –xv в строке #!/bin/sh, т.е.

```
#!/bin/sh -xv
```

Лучше всего применять ее совместно с перенаправлением вывода команде more или less**,** а также обоих потоков STDOUT и STDERR в определенный файл. Это позволит просмотреть и вывод самого сценария, и сообщения об ошибках:

```
#!/bin/sh -xv
# Пример возможностей трассировки в сценарии
# командного интерпретатора
result='echo "2 * 12 / (2 + 2)' | bc
echo $result
exit 0
```

Для запуска программы и перенаправления потоков STDOUT и STDERR команде more**,** применяется следующая команда:

```
./xvtest 2>&1 | more
```

Пример вывода программы :

```
1. ./xvtest 2>$1 | more
2.#!/bin/sh -xv
```

```
3.# Пример возможностей трассировки в сценарии командного
    интерпретатора
4.result=’ echo "2 * 12 / ( 2 + 3 ) " | bc
5.+ echo 2 * 12 / (2 + 3)
6.+ bc
7.+ result=4
8. echo $result
9.+ echo 4
10. 4
11. exit 0
12.+ exit 0
```

Здесь можно увидеть все действия, выполненные программой. Строки со знаком + представляют собой результаты этих действий. Например, в строке 3 результат вычислений присваивается переменной result. Предпринятые действия показаны в строках 4, 5 и 6. В строке 4 - видим команду echo**,** в строке 5 — запуск bc и, наконец, в строке 6 — присвоение результата (4) переменной result**.**

В этом же примере показано, как работает "раскрытие" переменных. Обратите внимание на строки 8—10. В строке 8 интерпретатор читает выражение с оператором echo**.** В строке 9 раскрывается значение переменной, в результате чего выражение превращается в "echo 4". В строке 10 печатается реальный вывод оператора echo.

2.4 Порядок выполнения

1) Использовать утилиту crash.

2) Использовать команду grep, например:

```
if grep user /etc/passwd>/dev/null 2>&1
then
echo пользователь user найден в файле паролей
fi
```

3) Создайте и запустите скрипт, фрагмент которого приведен ниже:

```
 if [ ! -f $HOME/.profile ]
 then
        echo "файла нет - скопируем шаблон"
        cp   /usr/lib/mkuser/sh/profile   $HOME/.profile
fi
```

4) Создайте и запустите скрипт, фрагмент которого приведен ниже:

```
    for   dir   in    /tmp   /usr/tmp  /home/tmp
   do
        find $dir !  -type  d  -atime  +11  -exec rm {} \;
done
```

5) Выполните и внесите в отчет результат выполнения данной последовательности команд:

```
mkdir testdir
echo some data > testdir/testfile
ls -l testdir/testfile
cat testdir/testfile
chmod 0200 testdir/testfile
cat testdir/testfile
chmod 0644 testdir/testfile
chmod 0444 testdir
ls testdir
cat testdir/testfile
chmod 0544 testdir/testfile
rm testdir/testfile
chmod 755 testdir
rm testdir/testfile
rm testfile
```

6) Отладить следующие скрипты:

а) проверка условия - пока i меньше или равно 20

```
#!/bin/sh
```

```
#  Count from 1 to 20
i=1
while [ $i -le 20 ]
do
        echo $i
        i='expr  $i +1'             i=$ ($i+1)
done
exit 0
```

б) цикл выполняется до тех пор, пока значение переменной i не станет больше 20

```
#!/bin/sh
# Count from 1 to 20
i=1
until [ $i -gt 20 ]
do
        echo $i
        i='expr $ +1'
done
exit 0
```

7) Отладить следующие скрипты:

```
#!/bin/sh
# Пример логического оператора AND
# В данном примере echo не запускается, т.к. VarA =1,  VarB не больше 7,
#поэтому оба условия не истинны, тест не проходит и возвращает значение
#0.
VarA=1
VarB=5
while [ $VarA -eg 1 ]  &&  [ $VarB -gt 7 ]
#сокр запись команды под названием test (-eg , истина , если 1 = 2
операнду.
```

```
do
        echo "VarA is equal to 1 and VarB is greater than 7"
done
exit 0

#!/bin/sh
# Пример логического оператора OR
#В данном случае достаточно, чтобы одно из условий было истинно.
#Выполняется бесконечный цикл с оператором echo
 /#Прервать - CTRL-C
VarA=1
VarB=5
while [ $VarA -eg 1 ]  ||  [ $VarB -gt 7 ]
do
        echo "VarA is equal to 1 and VarB is greater than 7"
done
exit 0
```

8) Отладить следующий сценарий:

```
#!/bin/sh
# Вывод квадратных корней чисел 10-20
for num in ' jot 10 10 20'
do
        square_root='echo "scale=5; sqrt($num)" | bc -lr'
echo $square_root
done
exit 0
```

Утилита jot печатает строку из 10 чисел, начиная с 10 и заканчивая 20 (10 10 20), scale=5 указывает bc, что в выводе числа после десятичной точки следует сохранить 5 значащих цифр. Знак ; используется для разделения

операторов, содержащихся в одной строке. Функция sqrt команды bc возвращает квадратный корень из заданного числа, т.к. интерпретатор заменяет переменную ее значением, bc получает число, а не имя переменной. Далее вывод команды перенаправляется команде bc. Опция -l указывает на необходимость предварительной загрузки математической библиотеки, где содержится функция sqrt.

9) Отладить сценарий программы по угадыванию числа.

```
#!/bin/sh
# Игра по угадыванию чисел
clear
guess_count=1    #Счетчик инициализируется значением 1
echo
echo "Number  guessing game written in bourne shell script"
echo
echo -n "Enter upper limit for guess: "
read up_limit
rnd_number='jot - r 1 1 $ up_limitr'    # Получение случайного числа
echo
echo  "Irve thought of a number between 1 and $up_limit."
echo
echo - n "Please guess a  number between 1 and $up_limit:  "
read guess
# Сравнение со случайным числом
while  true                    #строка 17
do
if  [ $guess - gt $rnd_number ]       #строка 19
then
echo
echo "Your guess was too high/ Please try again."
guess_count='expr $guess_count + 1'
```

```
echo - n "Please guess a number between 1 and $up_limit: "
read guess
elif  [ $guess -lt $rnd_number ]
then
        echo
        echo "Your guess was too low/ Please try again."
        guess_count='expr $guess_count + 1'
        echo - n "Please guess a number between 1 and $up_limit:  "
        read guess
else
        break
fi
done
# Мы достигаем этой точки, если игрок отгадал число.
echo
echo  "Correct!"
echo
echo  "yon guessed the number in $guess_count guesses."
echo
exit 0
```

Переменная guess_count хранит число попыток, сделанных игроком.

Команда jot выполняется для присвоения переменной rnd_number случайного числа в диапазоне от 1 до верхнего предела, заданного пользователем. Переменная guess сохраняет число, введенное игроком.

Оператор elif проверяет два два или более условий (else if). Сначала выполняется оператор if, если условие истинно, выполняется его код, затем управление передается следующему оператору. В строке 17 начинается бесконечный цикл. В строке 19 проверяется условие: число, введенное игроком больше, чем случайное число, выбранное программой.

Если да, игроку выводится сообщение, значение переменной guess_count увеличивается на 1, а затем запрашивается новое число. После того, как игрок вводит новое число, оно сохраняется в переменной guess и цикл while выполняется вновь. Если введенное число не больше случайного числа, оператор elif проверяет, меньше ли оно. Если да, игроку выводится сообщение, значение переменной guess_count увеличивается на 1, а затем запрашивается новое число. После ввода цикл while выполняется вновь. И наконец, если ни одно из условий не истинно, запускается блок else (здесь не нужны проверки, ведь если число ни больше и ни меньше, следовательно, оно равно загаданному). Оператор else прерывает бесконечный цикл, и программа передает управление первому оператору, следующему за оператором done. Последние выражения программы сообщают пользователю о том, что он угадал число, и о количестве предпринятых попыток (для этого выводится значение переменной guess_count).

2.5 Вопросы к защите лабораторной работы

1) Назначение и возможности языка командного интерпретатора Bourne Shell.
2) Что такое сценарий и для чего он предназначен?
3) Перечислите базовые классы и типы прав доступа к файлу.
4) Какие подстановки выполняет командный интерпретатор?
5) Использование условных операторов в скриптах.
6) Укажите различные форматы команды организации циклических операций.
7) Использование логических операторов AND/OR.
8) Использование функций в сценариях.
9) Для чего используются файловые дескрипторы?
10) Как выполнить отладку сценария командного интерпретатора?
11) Поясните на примере, как выполняется перехват прерываний?

3 ЛАБОРАТОРНАЯ РАБОТА №4 ПРОГРАММИРОВАНИЕ НА С В UNIX/LINUX

3.1 Цель работы

Целью данной работы является получение основных сведений о системных вызовах, базовых средств взаимодействия процессов, (порождение процесса, замена тела процесса, взаимодействие при помощи передачи/приема сигналов), отладка программы на С в UNIX/LINUX (анализ и исправление ошибок на стадии компиляции и в процессе работы программы).

3.2 Задание на выполнение работы

1) Установить компилятор GNU C Compiler под UNIX/LINUX*.

2) Написать и отладить программу, выводящую на экран строку «Hello UNIX»

- с использованием стандартной библиотеки ввода-вывода;
- с использованием функции непосредственной записи в стандартный поток вывода.

3) Написать и отладить программу, запрашивающую 2 числа и выводящую их наибольший общий делитель. Использовать алгоритм Евклида.

4) Написать и отладить программу, принимающую из командной строки 2 числа. Если эти числа равны, вывести на экран только одно число, если нет – вывести числа в порядке возрастания.

5) Написать и отладить программу, определяющую является ли файл исполняемым файлом формата ELF. Программа должна запросить у пользователя имя файла. Если файл не удастся открыть, проанализировать ошибку и выдать соответствующее сообщение.

6) Написать пример программы, которая запускает и связывает каналом два процесса: вывод содержимого каталога и подсчет количества строк (ls и wc).

7) Написать пользовательскую функцию обработки сигнала. Установка обработки сигнала происходит одноразово (обрабатывается только одно событие, связанное с появлением данного сигнала SIG_ALRM). Возврат из функции-обработчика происходит в точку прерывания процесса.

3.3 **Краткие теоретические сведения**

3.3.1 Установка компилятора

Компилятор GNU C Compiler представляет собой набор исполняемых файлов и библиотек. В дистрибутиве ASPLinux 7.3 он находится в пакете gcc-2.96-112asp.i386.rpm на диске 2. Для того чтобы поставить любой rpm-пакет в консоли надо набрать команду

rpm -i [имя_пакета] ,

в KDE следует использовать программу KPackager. Следует помнить, что между пакетами может существовать зависимость, т.е. один пакет не возможно поставить без предварительной установки другого. В случае с ASPLinux 7.3, при использовании типичной комплектации, при установке операционной системы gcc не ставится.

3.3.2 Компиляция

Процедура создания большинства приложений является общей. Первой фазой является стадия компиляции, когда файлы с исходными текстами программы, включая файлы заголовков, обрабатываются компилятором *cc*. Параметры компиляции задаются либо с помощью файла *makefile*, либо явным указанием необходимых опций компилятора в командной строке. В итоге компилятор создает набор промежуточных объектных файлов. Традиционно имена созданных объектных файлов имеют суффикс « .o».

На следующей стадии эти файлы с помощью редактора связей *ld* связываются друг с другом и с различными библиотеками, включая стандартную библиотеку по умолчанию и библиотеки, указанные пользователем в качестве параметров. При этом редактор связей может выполняться в двух режимах: статическом и динамическом, что задается соответствующими опциями. В статическом, наиболее традиционном режиме связываются все объектные модули и статические библиотеки (их имена имеют суффикс «.a»), производится разрешение всех внешних ссылок модулей и создается единый исполняемый файл, содержащий весь необходимый для выполнения код. Во втором случае, редактор связей по возможности подключает разделяемые библиотеки (имена этих библиотек имеют суффикс «.so»). В результате создается исполняемый файл, к которому в процессе запуска на выполнение будут подключены все разделяемые объекты. В обоих случаях по умолчанию создается исполняемый файл с именем *a.out.*

В UNIX/LINUX при использовании компилятора gcc исходник программы пишется в обычном текстовом редакторе (например встроенный в *mc* редактор) и сохраняется с расширением .c. Далее исходник транслируется:

$ cc -c [имя__исходного_файла]

после данной команды получается объектный файл с расширением .o (если не было ошибок) и компилируется:

$ *cc -o [имя_исполняемого_файла] [имя_объектного файла]*

после данной команды получается исполняемый файл.

Для достаточно простых задач все фазы автоматически выполняются вызовом команды:

$ *make [имя исходного файла без расширения]*

Заметим, что команда *cc* является программной оболочкой и компилятора, и редактора связей, которую и рекомендуется использовать при создании программ.

3.3.3 Форматы исполняемых файлов

В большинстве современных операционных систем UNIX используются два стандартных формата исполняемых файлов – COFF (Common Object File Format) и ELF (Executable and Linking Format).

Описание форматов исполняемых файлов необходимо для описания базовой функциональности ядра операционной системы. Информация, хранящаяся в исполняемых файлах форматах COFF и ELF позволяет получить информацию для работы приложения и системы в целом: какие части программы необходимо загрузить в память; как создается область неинициализированных данных; где в памяти располагаются инструкции и данные программы, какие библиотеки необходимы для выполнения программы; как связан исполняемый файл на диске; образ программы в памяти и дисковая область свопинга.

Базовая структура памяти для процессов, загруженных из исполняемых файлов форматов COFF и ELF содержит одни и те же основные компоненты (сегменты кода, данных, стека), хотя расположение сегментов различно. Независимо от формата исполняемого файла виртуальные адреса процесса не могут выходить за пределы 3 Гбайт.

Формат ELF

Формат ELF имеет файлы нескольких типов. Стандарт ELF различает следующие типы:

1.*Перемещаемый файл* (relocatable file), хранящий инструкции и данные, которые могут быть связаны с другими объектными файлами. Результатом такого связывания может быть исполняемый файл или разделяемый объектный файл.

2.*Разделяемый объектный файл* (shared object file) также содержит инструкции и данные, но может быть использован двумя способами. В первом случае, он может быть связан с другими перемещаемыми файлами и разделяемыми объектными файлами, в результате чего будет создан новый объектный файл. Во втором случае, при запуске программы на выполнение операционная система может динамически связать его с исполняемым файлом программы, в результате чего будет создан исполняемый образ программы. В последнем случае речь идет о разделяемых библиотеках.

3.*Исполняемый файл* хранит полное описание, позволяющее системе создать образ процесса. Он содержит инструкции, данные, описание необходимых разделяемых объектных файлов, а также необходимую символьную и отладочную информацию.

ELF-файл

Заголовок ELF-файла имеет фиксированное расположение. Остальные компоненты размещаются в соответствии с информацией, хранящейся в заголовке. Таким образом, заголовок содержит общее описание структуры файла, расположение отдельных компонентов и их размеры. Поскольку заголовок ELF-файла определяет его структуру, в таблице 3.1 приведены более подробно поля заголовка.

Таблица 3.1 –Поля заголовка ELF-файла

Поле	Описание
1	2
e_ident [ ]	Массив байт, каждый из которых определяет некоторую общую характеристику файла: формат файла (ELF), номер версии, архитектуру „ системы (32-разрядная или 64-разрядная) и т. д.
e_type	Тип файла
e_machine	Архитектура аппаратной платформы, для которой создан данный файл
e_version	Номер версии ELF-формата. Обычно определяется как EV_CURRENC (текущая), что означает последнюю версию
e_entry	Виртуальный адрес, по которому системой будет передано управление после загрузки программы (точка входа)
e_phoff Р	Расположение (смещение от начала файла) таблицы заголовков

	программы
e_shof f	Расположение таблицы заголовков секций
1	2
e_ehsize	Размер заголовка
e_phentsize	Размер каждого заголовка программы
e_phnum	Число заголовков программы
e_shentsize	Размер каждого заголовка сегмента (секции)
e_shnum	Число заголовков сегментов (секций)
e_shstrndx	Расположение сегмента, содержащего таблицу строк

Информация, содержащаяся в таблице заголовков программы, указывает ядру, как создать образ процесса из сегментов. Большинство сегментов копируются (отображаются) в память и представляют собой соответствующие сегменты процесса при его выполнении, например, сегменты кода или данных.

Каждый заголовок сегмента программы описывает один сегмент и содержит следующую информацию:

- тип сегмента и действия операционной системы с данным сегментом;
- расположение сегмента в файле;
- стартовый адрес сегмента в виртуальной памяти процесса;
- размер сегмента в файле;
- размер сегмента в памяти;
- флаги доступа к сегменту (запись, чтение, выполнение).

Часть сегментов имеет тип LOAD, предписывающий ядру при запуске программы на выполнение создать соответствующие этим сегментам структуры данных, называемые *областями,* определяющие непрерывные участки виртуальной памяти процесса и связанные с ними атрибуты. Сегмент, расположение которого в ELF-файле указано в соответствующем заголовке программы, будет отображен в созданную область, виртуальный адрес начала которой также указан в заголовке программы. К сегментам такого типа относятся, например, сегменты, содержащие инструкции программы (код) и ее данные. Если размер

сегмента меньше размера области, неиспользованное пространство может быть заполнено нулями. Такой механизм, в частности используется при создании неинициализированных данных процесса (BSS).

В сегменте типа INTERP хранится программный интерпретатор. Данный тип сегмента используется для программ, которым необходимо динамическое связывание. Суть динамического связывания заключается в том, что отдельные компоненты исполняемого файла (разделяемые объектные файлы) подключаются не на этапе компиляции, а на этапе запуска программы на выполнение. Имя файла, являющегося *динамическим редактором связей,* хранится в данном сегменте. В процессе запуска программы на выполнение ядро создает образ процесса, используя указанный редактор связей. Таким образом, первоначально в память загружается не исходная программа, а динамический редактор связей. На следующем этапе динамический редактор связей совместно с ядром UNIX создают полный образ исполняемого файла. Динамический редактор загружает необходимые разделяемые объектные файлы, имена которых хранятся в отдельных сегментах исходного исполняемого файла, и производит требуемое размещение и связывание. В заключение управление передается исходной программе.

Завершает файл таблица заголовков *разделов* или *секций* (section). Разделы (секций) определяют разделы файла, используемые для связывания с другими модулями в процессе компиляции или при динамическом связывании. Соответственно, заголовки содержат всю необходимую информацию для описания этих разделов. Как правило, разделы содержат более детальную информацию о сегментах. Так, например, сегмент кода может состоять из нескольких разделов, таких как хэш-таблица для хранения индексов используемых в программе символов, раздел инициализационного кода программы, таблица связывания, используемая динамическим редактором, а также раздел, содержащий собственно инструкции программы.

Следует обратить внимание на то, что в начале ELF-файла стоит сигнатура 'Ox7fELF' , по которой можно определить формат файла (массив байт, каждый из которых определяет некоторую общую характеристику файла: формат файла (ELF), номер версии, архитектуру системы (32-разрядная или 64-разрядная) и т. д.).

3.3.4 Утилита crash

Фактическую информацию о структурах управления адресным пространством процесса можно получить с помощью команды *crash().* В следующем примере определяется содержимое структур pregion процесса и характеристики соответствующих областей.

```
# crasn
dumpfile = /dev/mem, namelist = /unix, outfile = stdout
> pregion 101
  SLOT   PREG   REG#        REGVA    TYPE    FLAGS
   101      0     12     0x700000    text    rdonly
            1     22     0x701000    data
            2     23    0x7ffffffc   stack
            3    145    0x80001000   lbtxt   rdonly
            4    187    0x80031000   lbdat   pr
```

Как можно увидеть из вывода команды *crash(),* с рассматриваемым процессом связаны пять областей: сегмент кода, данных и стека, а также сегменты кода и данных подключенной библиотеки. Столбец REG# определяет запись таблицы областей, где расположена адресуемая каждой pregion область region. Заметим, что значение в столбце REG# лишь отчасти соответствует полю p_reg структуры pregion, поскольку последнее является указателем, а не индексом таблицы. Столбец REGVA содержит значения виртуальных адресов областей

С помощью полученной информации можно более детально рассмотреть любую из областей процесса. Выведем данные о сегментах кода, данных и стека:

```
$ region 12 22 23
SLOT PGSZ VALID SMEM NONE   SOFF REF  SWP NSW  FORM BACK INOX TYPE  FLAGS
 12    1     1    1    0     0   11    0   0    15    5   154  stxt  done
223    1     0    0    0     1   0     0   0    238   23  154  priv  done
```

```
23 2     1     1     0     0  1     0  0     135    24          priv    stack
```

Столбец PGSZ определяет размер области в страницах, а столбец VALID -число страниц этой области, находящихся в оперативной памяти. Как можно заметить, для сегментов данных и стека страниц недостаточно, поэтому может возникнуть ситуация, когда процессу потребуется обращение к адресу, в настоящее время отсутствующему в памяти. Столбец INOX содержит индексы таблиц inode, указывающие на метаданные файлов, откуда было загружено содержимое соответствующих сегментов.

Можно получить дополнительные сведения об этом файле:

$ inode 154

```
INODE TABLE SIZE = 472
SLOT  MAJ/MIN  FS  INUMB  RCNT  LINK  UID  GID  SIZE  MODE  MNT  M/ST  FLAGS
154   1,42     2   1562   3     1     123  56   8972  f---755  0  R130  tx
```

Из этой таблицы мы можем определить файловую систему, в которой расположен файл (MAJ/MIN), а также номер его дискового inode -INUMB. В данном случае он равен 1562. Выполнив команду *ncheck,* можно узнаем имя исполняемого файла, соответствующего исследуемому процессу:

```
$ ncheck   -i  1562
  /de/root:
1562     /home/andrei/CH3/test
```

3.3.5 Общая структура программы

Общая структура программы выглядит следующим образом:

```
#include <stdio.h>                    //стандартные заголовочные файлы
#include «my_file.h»                  //пользовательские заголовочные файлы
main (int argc, char* argv[])   //функция main и ее параметры
{
    printf(«Hello world!!!\n»);        //стандартные функции
    my_printf(«Hello_user!!!\n»);      //пользовательские функции
    return 0;
```

}

В приведенной структуре:

int argc - количество аргументов командной строки. Если программа запущена без аргументов argc = 1.

char argv[]* - массив указателей на аргументы командной строки, argv[0] - всегда имя программы.

3.3.6 Программные каналы

Для организации взаимодействие между несколькими процессами путем передачи данных от одного процесса к другому в системе UNIX используются каналы. С точки зрения программы, канал есть некая сущность, обладающая двумя файловыми дескрипторами (ФД). Через один ФД процесс может записать информацию в канал, через другой ФД процесс может читать информацию из канала. Так как канал это нечто, связанное с файловыми дескрипторами, то канал может передаваться по наследству дочерним процессам. Это означает, что два родственных процесса могут обладать одним и тем же каналом, т.е. если один процесс запишет какую-то информацию в канал, то другой процесс может прочесть эту информацию из этого же канала.

Функции дублирования файловых дескрипторов

int dup(fd); int dup2(fd, to_fd);

int fd; int fd, to_fd,

Аргументом функции dup является файловый дескриптор открытого в данном процессе файла. Эта функция возвращает -1 в том случае если обращение не проработало, и значение больше либо равное нулю, если работа функции успешно завершилась. Работа функции заключается в том, что осуществляется дублирование ФД в некоторый свободный ФД. Т.е. можно как бы продублировать открытый файл.

Функция dup2 дублирует файловый дескриптор fd в некоторый файловый дескриптор с номером to_fd. При этом, если при обращении к этой функции ФД в который мы хотим дублировать был занят, то

происходит закрытие файла, работающего с этим ФД, и переопределение ФД. Рассмотрим пример:

```
int   fd,
chars[80];
fd = open ("a.txt",O_RDONLY);
dup2 (fd,0);
close (fd);
gets (s,80);
```

Программа открывает файл с именем a.txt только на чтение. Файловый дескриптор, который будет связан с этим файлом, находится в fd. Далее программа обращается к функции dup2, в результате чего будет заменен стандартный ввод процесса на работу с файлом a.txt. Далее можно закрыть дескриптор fd. Функция gets прочтет очередную строку из файла a.txt.

Особенности работы с каналом

Под хранение информации передаваемой через канал выделяется некоторый фиксированный объем оперативной памяти. В некоторых системах этот буфер может быть продолжен на внешнюю память. Что происходит, если процесс хочет записать информацию в канал, но буфер переполнен, или прочесть информацию из канала, но в буфере нет еще данных? В обоих случаях процесс приостанавливает свое выполнение и дожидается, пока не освободится место либо, соответственно, пока в канале не появится информация. Надо заметить, что в этих случаях работа процесса может изменяться в зависимости от установленных параметров, которые можно менять программно. Для реализации каналов в системе используется функция pipe. Аргументом этой_функции должен быть указатель на массив двух целых переменных:

```
int pipe (pipes);
int pipes[2];
```

Нулевой элемент массива после обращения к функции pipe получает ФД для чтения, первый элемент этого массива получает ФД для записи. Если нет свободных ФД, то эта функция возвращает -1. Признак конца файла для считывающего дескриптора не будет получен до тех пор, пока не закрыты все дескрипторы, связанные с записью в этот канал.

Рассмотрим небольшой пример:

```
char *s = "Это пример";
char  b[80];
int  pipes[2];
pipe (pipes);
write (pipes[l],s, strlen(s)+l);
read (pipes[0],s, strlen(s)+l),
```

Это пример копирования строки (понятно, что так копировать строки не надо, и вообще никто функцией pipe в пределах одного процесса не пользуется). В этом примере и в последующих не обрабатываются случаи отказа.

3.3.7 Сигналы

Ввзаимодействие между процессами можно осуществить с помощью приема-передачи сигналов. В системе Unix можно построить аналогию механизму прерываний из некоторых событий, которые могут возникать при работе процессов.

Эти события, также как прерывания, однозначно определены для конкретной версии ОС, т.е. определен набор сигналов. Возникновение сигналов, почти также как и возникновение прерываний может происходить по следующим причинам:

- некоторое событие внутри программы, например, деление на ноль или переполнение;
- событие, связанное с приходом некоторой информации от устройства, например, событие, связанное с передачей от клавиатуры комбинации "Ctrl+C";

- событие, связанное с воздействием одного процесса на другой, например, "SIG_KILL".

Система имеет фиксированный набор событий, которые могут возникать. Каждое событие имеет свое уникальное имя, эти имена обычно едины для всех версий Unix. Такие имена называются сигналами. Перечень сигналов находится в include-файле "signal.h".

Прототип функции обработки сигнала:

void (* signal (sig, fun)) ()

int sig;

void (* fun) ();

При обращении к signal передаем: sig — имя сигнала; fun — указатель на функцию, которая будет обрабатывать событие, связанное с возникновением этого сигнала. Она возвращает указатель на предыдущую функцию обработки данного сигнала.

Событие, связанное с возникновением сигнала может быть обработано в системе тремя способами:

SIG_DEF — стандартная реакция на сигнал, которая предусмотрена системой;

SIG_IGN — игнорирование сигнала (следует отметить, что далеко не все сигналы можно игнорировать, например, SIG_KILL).

Еще две функции, которые необходимы для организации взаимодействия между процессами:

- int kifl(int pid, sig) — это функция передачи сигнала процессу. Она работает следующим образом: процессу с номером pid осуществляется попытка передачи сигнала, значение которого равно sig. Соответственно, сигнал может быть передан в рамках процессов, принадлежащих одной группе. Код ответа: -1, если сигнал передать не удалось. Функция kill может использоваться для проверки существования процесса с заданным идентификатором. Если функция выполняется с sig=0, то это тестовый сигнал, который

определяет - можно или не тпередать процессу сигнал, если можно, то код ответа kill отличен от "-1". Если pid=0, то заданный сигнал передается всем процессам, входящим в группу.

- int wait(int *wail_rel). Ожидание события в дочернем процессе. Если его нет, то управление возвращается сразу же с кодом ответа "-1". Если в дочернем процессе возникло событие, то анализируются младшие 16 бит в значении wait_ret:

  - если дочерний процесс приостановлен (трассировка или получение сигнала), тогда старшие 8 бит wait_ret - код сигнала, который получил дочерний процесс, а младшие содержат код 0177.

  - если дочерний процесс успешно завершился через обращение к функции exit. Тогда младшие 8 бит равны нулю, а старшие 8 бит равны коду, установленному функцией exit.

  - если дочерний процесс завершился из-за возникновения у него необрабатываемого сигнала, то старшие 8 бит равны нулю, а младшие - номер сигнала, который завершил процесс.

Функция wait возвращает идентификатор процесса в случае успешного выполнения и "-1" в противном случае. Если одно из перечисленных событий произошло до обращения к функции, то результат возвращается сразу же, то есть никакого ожидания не происходит, это говорит о том, что информация о событиях в процессе безвозвратно не теряется.

Рассмотрим пример многопроцессного взаимодействия.

```
alr()
{
 printf ("\n Быстрее!!! \n");
signal (SIG_ALRM, alr);
}
main ()
{
char s[80];
```

```
 int pid;
signal(SIG_ALRM, alr);
if (pid=fork())
        for (;;)
                {
                sleep(5); kill(pid, SIG_ALRM); /* приостанавливаем процесс на 5 секунд
                                                  и отправляем сигнал SIG_ALRM
                                                   дочернему процессу  */
                }
                printf ("имя?");
                for (;;)
                 {
                printf ("имя?");
                if gets(s,80)!=NULL) break;
                 }
        printf("OK!\n");
kill(getpid(), SIG_KILL);           /* убиваем зациклившийся родительский процесс */
}
```

Следует заметить, что в разных версиях Unix имена сигналов могут различаться.

3.4 Порядок выполнения

1) * Выполняется под контролем преподавателя.

2) Вывод на экран «Hello world!!!»

2.1
```
#include  <stdio.h>
        main ()
        {
        printf («Hello world!!!\n»);
        return 0;
        }
```

2.2
```
#include  <stdio.h>
```

```
main()
{
write(1,"Hello world!!!\n", 15);
return 0;
 }
```

3) Нахождение наибольшего общего делителя двух чисел по алгоритму Евклида. Алгоритм Евклида по нахождению наибольшего общего делителя следующий: пусть есть два числа *m и n* , *m* > *n*. Делим *m* на *n*, *r*-остаток от деления. Если остаток от деления равен нулю, то *n*-наибольший общий делитель. Если *n* не равен 0, заменяем *m* на *n* и *n* на *r*. Продолжаем деление, пока *r* не станет равным 0.

```
#include  <stdio.h>
unsigned int m,n,r;
main()
{
      scanf("%u   %u",  &m,  &n);
      if (n > m)
      {
            r=m;
            m=n;
            n=r;
      }
      printf ("Наибольший общий делитель %u и %u:  ",m,n);
      while  (1)
      {
            r=m%n;
            if ( r = = 0)
          break;
            else
            {
```

```
            m=n;
            n=r;
             }
        }
        printf ("%u\n",n); r
        return 0;
}
```

4) Работа с аргументами командной строки

```
  #include  <stdio.h>
#include  <string.h>
#include  <stdlib.h>
main  (int argc, char* argv[])
{
        if (argc < 3)
        {
        printf ("Недостаточно параметров\n");
         return 0;
        }
        if  (!strcmp(argv[l],argv[2]))
           printf ("%s\n",argv[l] );
        else
            atoi(argv[l]) < atoi(argv[2]) ? printf ("%s%s\n", argv[l], argv[2]) :
            printf ("%s    "%s \n", argv[2] , argv[l] ) ;
        return 0;
}
```

5) Определение формата файла

```
#include  <stdio.h>
#include  <errno.h>
 unsigned char szSig[]={'\x7F','\x45','\x4C','\x46'};
 unsigned char szBuffer[4];
```

```
char  szFileName[20];
FILE*  hFile;
main()
{
   scanf("%s",szFileName);
   hFile=fopen (szFileName, "r");
   if (hFile ==NULL)
     {
          switch(errno)
           {
                case 2:
                     printf  ("Файл не найден\n");
                     break;
                case 13:
                     printf  ("Нет фоступа к файлу\n") ;
                     break;
                default:
                     printf  ("Неизвестная ошибка\n") ;
           }
          return 1;
          }
fread (szBuffer,1,4,hFile);
if  (!memcmp(szSig,szBuffer, 4))
     printf("Файл %s является исполняемым файлом формата ELF\n"
     ,szFileName);
 else
     printf("Файл %s не является исполняемым файлом формата ELF\n "
     ,szFileName);
fclose(hFile);
return 0;
```

```
}
```

6) Организация каналов

```
main ()
{
int fd[2];
pipe (fd);        /*в родительском  процессе образуем два дескриптора канала
*/
if (fork())    /* образуем дочерний процесс, у которого будут те же
               дескрипторы */
{                /* эта часть программы происходит в родительском
               процессе*/
dup2(fd[1],1);       /* заменяем стандартный вывод выводом в канал */
close(fd[1]);        /*закрываем дескрипторы канала*/ ,
close(fd[0]);        /* теперь весь вывод будет происходить в канал */
exec1("/bin/ls", "ls", (char*)0); /* заменяем тело родителя на ls */
}                    /* отсюда начинает работать дочерний процесс */
dup2(fd[0],0),       /* в дочернем процессе  все делаем аналогично */
close(fd[0]);
close(fd[1]);
exe1l("/bin/wc", "wc", (char*)0);
}
```

В отцовском процессе запущен процесс ls. Всю выходную информацию ls загружает в канал (ассоциировали стандартное устройство вывода с каналом). Далее в дочернем запустили процесс wc, у которого стандартное устройство ввода связано с дескриптором чтения из канала. Это означает, что все то, что будет писать ls в свое стандартное устройство вывода, будет поступать на стандартное устройство ввода команды wc.

Для того, чтобы канал работал корректно, и читающий дескриптор получил признак конца файла, должны быть закрыты все пишущие дескрипторы. Если в программе не была бы указана выделенная строка, то

процесс, связанный с wc завис, так как в этом случае функция, читающая из канала, не дождется признака конца файла. Она будет ожидать его бесконечно долго. В родительском процессе подчеркнутую строку можно было бы не указывать, т.к. дескриптор закрылся бы при завершении процесса, а в дочернем процессе такая строка нужна.

7) Пример программы "Будильник". Функция alrm инициализирует появление сигнала SIG_ALRM

```
main ()
{
chars[80];
signal (SIG_ALRM, alrm); /* установка режима связи с событием SIG_ALRM на
                            функцию alrm */
alarm(5);                /* заводим будильник */
printf ("Введите имя \n");
for (;;)
    {
   printf{"имя:");
if (gets(s,80) != NULL) break;
    }
printft ("OK! \n");
}
alrm()
{
    printf ("\n жду имя \n");
alarm(5);
signal (SlG_ALRM, alrm);
}
```

В начале программы устанавливаем реакцию на сигнал SIG_ALRM на функцию alrm, далее заводим будильник, запрашиваем "Введите имя" и ожидаем ввода строки символов. Если ввод строки задерживается, то будет

вызвана функция alrm, которая напомнит, что программа "ждет имя", опять заведет будильник и поставит себя на обработку сигнала SIG_ALRM еще раз. И так будет до тех пор, пока не будет введена строка.

Если в момент выполнения системного вызова возникает событие, связанное с сигналом, то система прерывает выполнение системного вызова и возвращает код ответа, равный "-1". Это можно также проанализировать по функции errno.

3.5 Вопросы к защите лабораторной работы

1) Опишите схему компиляции программы.
2) Перечислите форматы исполняемых файлов.
3) Перечислите этапы выполнения программы в операционной системе UNIX.
4) Перечислите основные системные функции для работы с файлами.
5) Перечислите основные интерфейсы для файлового ввода/вывода.
6) Перечислите базовые средства взаимодействия процессов в UNIX.
7) Поясните особенности работы с каналами в UNIX.
8) Почему отложенные вызовы не обрабатываются непосредственно обработчиком прерывания таймера?
9) Как получить данные о сегментах кода, данных и стека?

4 Литература

1) А. Робачевский Операционная система UNIX.-СПб.:BHV-Санкт-Петербург, 2002- 528 с.
2) FreeBSD/ Энциклопедия пользователя: Пер. с англ./ Майкл Эбен, Брайан Таймэн – К.: ООО ТИД ДС, 2002 - 736.
3) Ю. Вахалия UNIX изнутри. – СПб.: Питер, 2003 – 844 с.