Практикум по операционным системам

Практикум по операционным системам состоит из трех частей:
1. Теоретический минимум и задания для тренировки
2. Базовый набор заданий (обязательный для всех)
3. Дополнительный набор заданий

Рекомендуется выполнять задания в виртуальной машине с Linux CentOS. Сборка под Oracle VirtualBox доступна по ссылке: https://disk.yandex.ru/d/d48ND4ylz_-0pA
В виртуальной машине создан пользователь user с паролем 123654. Пароль для пользователя root задан как rootroot.
При работе под суперпользователем рекомендуется создавать снэпшоты виртуальной машины.
Если есть опытные пользователи Linux, можно использовать любые дистрибутивы, при необходимости, внося корректировки в формулировки заданий.
Защита заданий заключается в демонстрации преподавателю полученных результатов и ответов на вопросы по коду или выполнение небольшого дополнительного задания. В случае успешной защиты начисляются баллы за конкретное задание.

Часть 1. Теоретический минимум и задания для тренировки

Раздел 1. Основы использования консольного интерфейса ОС Linux и интерпретатора bash.

Для получения подробного **справочного руководства** по любой команде можно набрать в консоли «man название команды», для кратной справки – название_команды --help.
Примеры: *man man* – справочное руководство по команде man; *man bash* – справочное руководство по интерпретатору bash.

Shell-скрипт – это обычный текстовый файл, в который последовательно записаны команды, которые пользователь может обычно вводить в командной строке. Файл выполняется командным интерпретатором – шеллом (shell). В Linux- и Unix-системах для того, чтобы бинарный файл или скрипт смогли быть запущены на выполнение, для пользователя, который запускает файл, должны быть установлены соответствующие права на выполнение. Это можно сделать с помощью команды `chmod u+x имя_скрипта`. В первой строке скрипта указывается путь к интерпретатору с помощью конструкции `#!/bin/bash`.
Для создания скрипта можно воспользоваться текстовым редактором `nano` или `vi`, набрав имя редактора в командной строке.

Ниже приводятся основные правила программирования на языке bash.
Комментарии. Строки, начинающиеся с символа # (за исключением комбинации `#!`), являются комментариями. Комментарии могут также располагаться и в конце строки с исполняемым кодом.
Особенности работы со строками. Одиночные кавычки (`' '`), ограничивающие подстроки с обеих сторон, служат для предотвращения интерпретации специальных символов, которые могут находиться в строке. Двойные кавычки (`" "`) предотвращают интерпретацию специальных символов, за исключением `$, ` (обратная кавычка) и \ (escape – обратный слэш). Желательно использовать двойные кавычки при обращении к переменным. При необходимости вывести специальный символ можно также использовать экранирование: символ \ предотвращает интерпретацию следующего за ним символа.
Переменные. Имя переменной аналогично традиционному представлению об идентификаторе, т.е. именем может быть последовательность букв, цифр и подчеркиваний, начинающаяся с буквы или подчеркивания. Когда интерпретатор встречает в тексте сценария имя переменной, то он вместо него подставляет значение этой переменной. Поэтому ссылки на переменные называются подстановкой переменных. Если `variable1` – это имя переменной, то `$variable1` – это ссылка на ее значение. "Чистые" имена переменных, без префикса `$`, могут использоваться только при объявлении переменной или при присваивании переменной некоторого значения. В отличие от большинства других языков программирования, Bash не производит разделения переменных по типам. По сути, переменные Bash являются строковыми переменными, но, в зависимости от контекста, Bash допускает целочисленную арифметику с переменными.
Оператор присваивания "=". При использовании оператора присваивания нельзя ставить пробелы слева и справа от знака равенства. Если в процессе присваивания требуется выполнить арифметические операции, то перед записью арифметического выражения используют оператор `let`, например:

```
Let a=2\*2
```

(оператор умножения является специальным символом и должен быть экранирован).
Арифметические операторы:

"+" сложение
"–" вычитание
"*" умножение
"/" деление (целочисленное)
"**" возведение в степень
"%" остаток от деления

Специальные переменные. Для Bash существует ряд зарезервированных имен переменных, которые хранят определенные значения.

– Позиционные параметры. Аргументы, передаваемые скрипту из командной строки, хранятся в зарезервированных переменных `$0, $1, $2, $3...`, где `$0` – это название файла сценария, `$1` – это первый аргумент, `$2` – второй, `$3` – третий и так далее. Аргументы, следующие за `$9`, должны заключаться в фигурные скобки, например: `${10}, ${11}, ${12}`. Передача параметров скрипту происходит в виде перечисления этих параметров после имени скрипта через пробел в момент его запуска.
– Некоторые другие зарезервированные переменные:
  `$HOME` – домашний каталог пользователя
  `$HOSTNAME` – hostname компьютера
  `$PWD` – рабочий каталог
  `$PATH` – путь поиска программ
  `$PPID` – идентификатор родительского процесса
  `$SECONDS` – время работы скрипта (в секундах)
  `$#` – общее количество параметров, переданных скрипту
  `$*` – все аргументы, переданные скрипту (выводятся в строку)
  `$@` – то же самое, что и предыдущий, но параметры выводятся в столбик
  `$!` – PID последнего запущенного в фоне процесса
  `$$` – PID самого скрипта

Код завершения. Команда `exit` может использоваться для завершения работы сценария, точно так же как и в программах на языке С. Кроме того, она может возвращать некоторое значение, которое может быть проанализировано вызывающим процессом. Команде `exit` можно явно указать код возврата, в виде `exit nnn`, где `nnn` – это код возврата (число в диапазоне 0– 255).

Оператор вывода. `Echo переменные_или_строки`

Оператор ввода. `Read имя_переменной.` Одна команда `read` может прочитать (присвоить) значения сразу для нескольких переменных. Если переменных в `read` больше, чем их введено (через пробелы), оставшимся присваивается пустая строка. Если передаваемых значений больше, чем переменных в команде `read`, то лишние игнорируются.

Условный оператор. `If команда; then команда; [else команда]; fi.`

Если команда вернула после выполнения значение "истина", то выполняется команда после `then`. Если есть необходимость сравнивать значения переменных и/или констант, после `if` используется специальная команда `[[ выражение ]].` Обязательно ставить пробелы между выражением и скобками, например:

```
if [[ "$a" -eq "$b" ]]
then echo "a = b"
fi
```

Операции сравнения:

Для строк
`-z` # строка пуста
`-n` # строка не пуста
`=, (==)` # строки равны
`!=` # строки не равны
`<` # меньше
`>` # больше

Для числовых значений
`-eq` # равно
`-ne` # не равно
`-lt` # меньше
`-le` # меньше или равно
`-gt` # больше
`-ge` # больше или равно

`!` # отрицание логического выражения
`-a, (&&)` # логическое «И»
`-o, (||)` # логическое «ИЛИ»

Множественный выбор. Для множественного выбора может применяться оператор `case`.

```
case переменная in
значение1 )
команда 1
 ;;
значение2 )
команда 2
 ;;
esac
```

Выбираемые значения обозначаются правой скобкой в конце значения. Разделитель ситуаций – `;;`

Цикл for. Существует два способа задания цикла for.

1. Стандартный – `for переменная in список_значений do команды done.` Например:

```
for i in 0 1 2 3
do
echo $i
done
```

2. С-подобный

```
for ((i=0; c <=3; i++))
do
echo $i
done
```

Цикл while: `while условие; do; команда; done.` Синтаксис записи условия такой же, как и в условном операторе.

Управление циклами. Для управления ходом выполнения цикла служат команды `break` и `continue`. Они точно соответствуют своим аналогам в других языках программирования. Команда `break` прерывает исполнение цикла, в то время как `continue` передает управление в начало цикла, минуя все последующие команды в теле цикла.

Управление вводом-выводом команд (процессов)

У любого процесса по умолчанию всегда открыты три файла – **`stdin`** (стандартный ввод, клавиатура), **`stdout`** (стандартный вывод, экран) и **`stderr`** (стандартный вывод сообщений об ошибках на экран). Эти и любые другие открытые файлы могут быть перенаправлены. В данном случае термин "перенаправление" означает: получить вывод из файла (команды, программы, сценария) и передать его на вход в другой файл (команду, программу, сценарий).

`команда > файл` – перенаправление стандартного вывода в файл, содержимое существующего файла удаляется.

`команда >> файл` – перенаправление стандартного вывода в файл, поток дописывается в конец файла.

`команда1 | команда2` – перенаправление стандартного вывода первой команды на стандартный ввод второй команды = образование конвейера команд.

`команда1 $(команда2)` – передача вывода команды 2 в качестве параметров при запуске команды 1. Внутри скрипта конструкция **`$(команда2)`** может использоваться, например, для передачи результатов работы команды 2 в параметры цикла **`for … in.`**

Работа с текстовыми потоками (внешние команды - утилиты)

Для каждой утилиты доступно управление с помощью передаваемых команде параметров. Рекомендуем ознакомиться с документацией по этим командам с помощью команды man.

`sort` – сортирует поток текста в порядке убывания или возрастания, в зависимости от заданных опций.

`uniq` – удаляет повторяющиеся строки из отсортированного файла.

`cut` – извлекает отдельные поля из текстовых файлов (поле – последовательность символов в строке до разделителя).

`head` – выводит начальные строки из файла на **`stdout.`**

`tail` – выводит последние строки из файла на **`stdout.`**

`wc` – подсчитывает количество слов/строк/символов в файле или в потоке

`tr` – заменяет одни символы на другие.

Полнофункциональные многоцелевые утилиты:

`grep` – многоцелевая поисковая утилита, использующая регулярные выражения.

`sed` – неинтерактивный "потоковый редактор". Принимает текст либо из **`stdin`**, либо из текстового файла, выполняет некоторые операции над строками и затем выводит результат в **`stdout`** или в файл. **`Sed`** определяет,

по заданному адресному пространству, над какими строками следует выполнить операции. Адресное пространство строк задается либо их порядковыми номерами, либо шаблоном. Например, команда **3d** заставит **sed** удалить третью строку, а команда **/windows/d** означает, что все строки, содержащие "**windows**", должны быть удалены. Наиболее часто используются команды **p** – печать (на **stdout**), **d** – удаление и **s** – замена.
awk – утилита контекстного поиска и преобразования текста, инструмент для извлечения и/или обработки полей (колонок) в структурированных текстовых файлах. **Awk** разбивает каждую строку на отдельные поля. По умолчанию поля – это последовательности символов, отделенные друг от друга пробелами, однако имеется возможность назначения других символов в качестве разделителя полей. **Awk** анализирует и обрабатывает каждое поле в отдельности.

Регулярные выражения – это набор символов и/или метасимволов, которые наделены особыми свойствами. Их основное назначение – поиск текста по шаблону и работа со строками. При построении регулярных выражений используются нижеследующие конструкции (в порядке убывания приоритета), некоторые из которых могут быть использованы только в расширенных версиях соответствующих команд (например, при запуске **grep** с ключом -**E**).

c Любой неспециальный символ **c** соответствует самому себе
\c Указание убрать любое специальное значение символа **c** (экранирование)
^ Начало строки
$** Конец строки; выражение **"^$" соответствует пустой строке.
. Любой одиночный символ, за исключением символа перевода строки
[...] Любой символ из **...;** допустимы диапазоны типа **a-z;** возможно объединение диапазонов, например **[a-z0-9]**
[^...] Любой символ не из **...;** допустимы диапазоны
r* Ноль или более вхождений символа **r** (может применяться и для диапазонов)
r+ Одно или более вхождений символа **r** (может применяться и для диапазонов)
r? Нуль или одно вхождение символа **r** (может применяться и для диапазонов)
\<...\> Границы слова
\{ \} Число вхождений предыдущего выражения. Например, выражение **"[0-9]\{5\}"** соответствует подстроке из пяти десятичных цифр
r1r2 За **r1** следует **r2**
r1|r2 **r1** или **r2**
(r) Регулярное выражение **r;** может быть вложенным

Классы символов POSIX

[:class:] альтернативный способ указания диапазона символов.
[:alnum:] соответствует алфавитным символам и цифрам. Эквивалентно выражению [**A-Za-z0-9**].
[:alpha:] соответствует символам алфавита. Эквивалентно выражению [**A-Za-z**].
[:blank:] соответствует символу пробела или символу табуляции.
[:digit:] соответствует набору десятичных цифр. Эквивалентно выражению [**0-9**].
[:lower:] соответствует набору алфавитных символов в нижнем регистре. Эквивалентно выражению [**a-z**].
[:space:] соответствует пробельным символам (пробел и горизонтальная табуляция).
[:upper:] соответствует набору символов алфавита в верхнем регистре. Эквивалентно выражению [**A-Z**].
[:xdigit:] соответствует набору шестнадцатиричных цифр. Эквивалентно выражению [**0-9A-Fa-f**].

Задания для тренировки

Напишите скрипты, решающие следующие задачи:

i) В параметрах при запуске скрипта передаются три целых числа. Вывести максимальное из них.
ii) Вывести на экран текстовое меню с четырьмя пунктами и ожидать ввода пользователем номера пункта. При вводе пользователем номера пункта меню происходит запуск редактора nano, редактора vi, браузера links или выход из меню. После выхода из редакторов или браузера должно опять выводиться текстовое меню и происходить ожидание ввода номера пункта. При выборе номера 4 (выход из меню), скрипт завершается с кодом 0.
iii) Создать файл **emails.lst**, в который вывести через запятую все адреса электронной почты, встречающиеся во всех файлах директории /**etc**.
iv) Вывести список пользователей системы с указанием их UID, отсортировав по UID. Сведения о пользователей хранятся в файле /**etc/passwd**. В каждой строке этого файла первое поле – имя пользователя, третье поле – UID. Разделитель – двоеточие.
v) Подсчитать общее количество строк в файлах, находящихся в директории /**var/log**/ и имеющих расширение **log**.

Раздел 2. Мониторинг процессов и ресурсов в ОС Linux

Идентификация процессов

Система идентифицирует процессы по уникальному номеру, называемому идентификатором процесса или **PID** (process ID).

Все процессы, работающие в системе GNU/Linux, организованы в виде дерева. Корнем этого дерева является **init** или корневой процесс **systemd** – процессы системного уровня, запускаемые во время загрузки. Для каждого процесса хранится идентификатор его родительского процесса (**PPID,** Parent Process ID). Для корневого процесса в качестве идентификатора родительского процесса указывается 0.

Получение общих сведений о запущенных процессах

Команда **ps** (сокращение от process status)

Запуск **ps** без аргументов покажет только те процессы, которые были запущены Вами и привязаны к используемому Вами терминалу.

Часто используемые параметры (указываются без "-"):

a – вывод процессов, запущенные всеми пользователями;

x – вывод процессов без управляющего терминала или с управляющим терминалом, но отличающимся от используемого Вами;

u – вывод для каждого из процессов имя запустившего его пользователя и времени запуска.

Обозначения состояний процессов (в колонке **STAT**)

R – процесс выполняется в данный момент или находится в состоянии готовности;

S – процесс ожидает события (прерываемое ожидание);

D – процесс ожидает ввода/вывода (непрырываемое ожидание);

I – простаивающий поток ядра;

Z – zombie-процесс;

T – процесс остановлен.

Команда **top**

top – программа, используемая для наблюдения за процессами в режиме реального времени. Полностью управляется с клавиатуры. Вы можете получить справку, нажав на клавишу **h**. Наиболее полезные команды для мониторинга процессов:

M – эта команда используется для сортировки процессов по объему занятой ими памяти (поле **%MEM**);

P – эта команда используется для сортировки процессов по занятому ими процессорному времени (поле **%CPU**). Это метод сортировки по умолчанию;

U – эта команда используется для вывода процессов заданного пользователя. **top** спросит у вас его имя. Вам необходимо ввести имя пользователя, а не его UID. Если вы не введете никакого имени, будут показаны все процессы;

i – по умолчанию выводятся все процессы, даже спящие. Эта команда обеспечивает вывод информации только о работающих в данный момент процессах (процессы, у которых поле **STAT** имеет значение **R**, Running). Повторное использование этой команды вернет Вас назад к списку всех процессов.

Получение детальных сведений о запущенных процессах

/proc – псевдо-файловая система, которая используется в качестве интерфейса к структурам данных в ядре. Большинство расположенных в ней файлов доступны только для чтения, но некоторые файлы позволяют изменять переменные ядра.

Каждому запущенному процессу соответствует подкаталог с именем, соответствующим идентификатору этого процесса (его **PID**). Каждый из этих подкаталогов содержит следующие псевдо-файлы и каталоги (указаны наиболее часто использующиеся для мониторинга процессов). **Внимание!** Часть из этих файлов доступна только в директориях процессов, запущенных от имени данного пользователя или при обращении от имени **root**.

cmdline – файл, содержащий полную командную строку запуска процесса.

cwd – ссылка на текущий рабочий каталог процесса.

environ – файл, содержащий окружение процесса. Записи в файле разделяются нулевыми символами, и в конце файла также может быть нулевой символ.

exe – символьная ссылка, содержащая фактическое полное имя выполняемого файла.

fd – подкаталог, содержащий одну запись на каждый файл, который в данный момент открыт процессом. Имя каждой такой записи соответствует номеру файлового дескриптора и является символьной ссылкой на реальный файл. Так, **0** – это стандартный ввод, **1** – стандартный вывод, **2** – стандартный вывод ошибок и т. д.

io – файл, содержащий сведения об объемах данных, прочитанных и записанных процессом в хранилище.

maps – файл, содержащий адреса областей памяти, которые используются программой в данный момент, и права доступа к ним.

sched – файл, содержащий значения переменных для каждого процесса, использующихся планировщиком **CFS** для принятия решения о выделении процессу процессорного времени. Например, **sum_exec_runtime**, это переменная, содержащая оценку суммарного времени выполнения процесса, а **nr_switches** – переменная, хранящая количество переключений контекста.

stat – машиночитаемая информация о процессе в виде набора полей;
status – предоставляет бóльшую часть информации из **stat** в более лёгком для прочтения формате.
statm – предоставляет информацию о состоянии памяти в страницах как единицах измерения. Список полей в файле:

size общий размер программы
resident размер резидентной части
shared количество разделяемых страниц
text текст (код)
lib библиотеки (не используется начиная с ядра 2.6)
data данные + стек
dt "грязные" (dirty) страницы (не используется начиная с ядра 2.6)

Обработка данных о процессах
Обработка данных о процессах проводится, как правило, в рамках организации конвейера команд обработки текстовых потоков с использованием ранее рассмотренных утилит: **grep, sed, awk, tr, sort, uniq, wc**.

Задания для тренировки
Напишите скрипты, решающие следующие задачи:

i) Вывести в файл список **PID** всех процессов, которые были запущены командами, расположенными в **/sbin/**
ii) Для всех зарегистрированных в данный момент в системе процессов определить среднее время непрерывного использования процессора (CPU_burst) и вывести в один файл строки **ProcessID=PID : Parent_ProcessID=PPID : Average_Running_Time=ART.** Значения **PPid** взять из файлов **status**, которые находятся в директориях с названиями, соответствующими **PID** процессов в **/proc**. Значения **ART** получить, разделив значение **sum_exec_runtime** на **nr_switches**, взятые из файлов **sched** в этих же директориях. Отсортировать эти строки по идентификаторам родительских процессов.
iii) В полученном на предыдущем шаге файле после каждой группы записей с одинаковым идентификатором родительского процесса вставить строку вида **Average_Running_Children_of_ParentID=N is M**, где **N** = **PPID**, а **M** – среднее, посчитанное из **ART** для всех процессов этого родителя.
iv) Используя псевдофайловую систему **/proc** найти процесс, которому выделено больше всего оперативной памяти. Сравнить результат с выводом команды **top**.

Раздел 3. Управление процессами в ОС Linux

Команды для управления процессами
(с подробным описанием возможностей и синтаксисом команд можно ознакомиться в документации, доступной по команде **man команда**)
kill – передает сигнал процессу. Сигнал может передаваться в виде его номера или символьного обозначения. По умолчанию (без указания сигнала) передает сигнал требования о завершении процесса завершения процесса (**TERM**). Идентификация процесса для команды **kill** производится по PID. Перечень системных сигналов, доступных в GNU/Linux, с указанием их номеров и символьных обозначений можно получить с помощью команды **kill -l**;
killall – работает аналогично команде kill, но для идентификации процесса использует его символьное имя, а не PID;
pidof – определяет PID процесса по его имени;
pgrep – определяет PID процессов с заданными характеристиками (например, запущенные конкретным пользователем);
pkill – позволяет отправить сигнал группе процессов с заданными характеристиками;
nice – запускает процесс с заданным значением приоритета. Уменьшение значения (повышение приоритета выполнения) может быть инициировано только пользователем root;
renice – изменяет значения приоритета для запущенного процесса. Уменьшение значения (повышение приоритета выполнения) может быть инициировано только пользователем root;
tail – не только выводит последние n строк из файла, но и позволяет организовать "слежение" за файлом – обнаруживать и выводить новые строки, появляющиеся в конце файла.
sleep – задает паузу в выполнении скрипта.

Организация взаимодействия двух процессов
Существует несколько вариантов организации взаимодействия процессов. Поскольку суть взаимодействия состоит в передаче данных и/или управления от одного процесса к другому, рассмотрим два распространенных варианта организации такого взаимодействия: передачу данных через именованный канал и передачу управления через сигнал.

Взаимодействие процессов через именованный канал

Именованный канал – специальный тип файла в Linux. Создается командой **mkfifo имя_файла**.

Взаимодействие с именованным каналом происходит обычными средствами для взаимодействия с файлами, но при этом такой файл не будет сохраняться на носителе, а представляет собой буфер в памяти для организации межпроцессного обмена данными.

Для демонстрации передачи информации через именованный канал рассмотрим два скрипта – «Генератор» и «Обработчик». Требуется считывать информацию с одной консоли с помощью процесса «Генератор» и выводить ее на экран другой консоли с помощью процесса «Обработчик», причем таким образом, чтобы считывание генератором строки «QUIT» приводило к завершению работы обработчика. Каждый скрипт запускается независимо в своей виртуальной консоли. Переключаясь между консолями, можно управлять скриптами и наблюдать результаты их работы.

Перед запуском скриптов создадим именованный канал с помощью команды **mkfifo pipe**

Генератор	Обработчик
#!/bin/bash while true; do read LINE echo $LINE > pipe done	#!/bin/bash (tail -f pipe) | while true; do read LINE; case $LINE in QUIT) echo "exit" killall tail exit ;; *) echo $LINE ;; esac done

Скрипт «Генератор» в бесконечном цикле считывает строки с консоли и записывает их в именованный канал **pipe**.

Скрипт «Обработчик» рассмотрим подробнее.

Команда **tail** позволяет считывать последние **n** строк из файла. Но один из наиболее распространенных вариантов ее использования – организация «слежения» за файлом. При использовании конструкции **tail -f** считывание из файла будет происходить только в случае добавления информации в этот файл. Поскольку необходимо передавать выход команды **tail** на вход скрипта «Обработчик», используем конструкцию **(команды)|** Круглые скобки позволяют запустить независимый подпроцесс (дочерний процесс) внутри родительского процесса «Обработчик», а оператор конвейера в конце позволит направить выход этого подпроцесса на вход родительского процесса. Таким образом, команда **read** в этом скрипте читает выход команды **tail.** Остальная часть скрипта основывается на конструкциях, изученных в предыдущих лабораторных работах, и не требует детального рассмотрения. Исключение составляет только команда **killall tail.** С ее помощью завершается вызванный в подпроцессе процесс **tail** перед завершением родительского процесса. Использование **killall** в этом случае используется для упрощения кода, но не всегда является корректным. Лучше определять PID конкретного процесса **tail,** вызванного в скрипте, и завершать его с помощью команды **kill.**

Взаимодействие процессов с помощью сигналов

Сигналы являются основной формой передачи управления от одного процесса к другому. В Linux используются 64 различных сигнала. Любой процесс при создании наследует от родительского процесса таблицу указателей на обработчики сигналов и, тем самым, готов принять любой из 64-х сигналов. Обработчики по-умолчанию, как правило, приводят к завершению процесса или, реже, игнорированию сигнала, но за исключение двух сигналов (**KILL** и **STOP**), обработчики могут быть заменены на другие. У большинства сигналов есть предопределенное назначение и ситуации, в которых они будут передаваться, поэтому изменение их обработчиков возможно, но рекомендуется только в рамках расширения функционала соответствующего обработчика. Такие сигналы считаются системными. Но есть два сигнала (**USR1** и **USR2**), для которых предполагается, что обработчики будут

создаваться разработчиком приложения и использоваться для передачи и обработки пользовательских, неспецифицированных сигналов (по умолчанию, их обработка предполагает завершение процесса). Для замены обработчиков сигналов в **sh** (**bash**) используется встроенная команда **trap** с форматом

trap action signal

Команде нужно передать два параметра: действие при получении сигнала и сигнал, для которого будет выполняться указанное действие. Обычно в качестве действия указывают вызов функции, описанной выше в коде скрипта.

С помощью команды **trap** можно не только задать обработчик для пользовательского сигнала, но и подменить обработчик для некоторых из системных сигналов (кроме тех, перехват которых запрещен). В этом случае обработка сигнала перейдет к указанному в **trap** обработчику.

Для демонстрации передачи управления от одного процесса к другому рассмотрим еще одну пару скриптов.

Генератор	Обработчик

#!/bin/bash
while true; do
read LINE
case $LINE in
STOP)
kill -USR1 $(cat .pid)
;;
*)
:
;;
esac
done
	#!/bin/bash
	echo $$ > .pid
	A=1
	MODE="rabota"
	usr1()
	{
	MODE="ostanov"
	}
	trap 'usr1' USR1
	while true; do
	case $MODE in
	"rabota")
	let A=$A+1
	echo $A
	;;
	"ostanov")
	echo "Stopped by SIGUSR1"
	exit
	;;
	esac
	sleep 1
	done

В этом случае скрипт «Генератор» будет в бесконечном цикле считывать строки с консоли и бездействовать (используется оператор **:**) для любой входной строки, кроме строки STOP, получив которую, он отправит пользовательский сигнал **USR1** процессу «Обработчик». Поскольку процесс «Генератор» должен знать PID процесса «Обработчик», передача этого идентификационного номера осуществляется через скрытый файл. В процессе «Обработчик» определение PID процесса производится с помощью системной переменной **$$.** Процесс «Обработчик» выводит на экран последовательность натуральных чисел до момента получения сигнала **USR1**. В этот момент запускается обработчик **usr1()**, который меняет значение переменной MODE. В результате на следующем шаге цикла будет выведено сообщение о прекращении работы в связи с появлением сигнала, и работа скрипта будет завершена.

Задание для тренировки

Напишите скрипт, решающий следующую задачу: процесс «Генератор» считывает строки с консоли, пока ему на вход не поступит строка **TERM**. В этом случае он посылает системный сигнал **SIGTERM** процессу обработчику. Процесс «Обработчик» (как и в примере, выводящий в бесконечном цикле натуральное число каждую секунду) должен перехватить системный сигнал **SIGTERM** и завершить работу, предварительно выведя сообщение о завершении работы по сигналу от другого процесса.

Часть 2. Базовый набор заданий

Напишите скрипты, решающие следующие задачи (**каждая верно решенная задача = 2 балла**):

i) В файле /var/log/anaconda/syslog найти и вывести самое часто встречающееся слово длиной не менее 4 букв. Слово должно состоять только из английских букв и отделяться пробелами или знаками препинания. Поиск слова должен быть не чувствителен к регистру, то есть слова word, Word и WORD считаются за одно слово.

ii) Вывести на экран **`PID`** процесса, запущенного последним (имеющим максимальное время запуска на момент запуска этого скрипта).

iii) Создайте три фоновых процесса, выполняющих одинаковый бесконечный цикл вычисления (например, перемножение двух чисел). Определите их PIDы. Создайте скрипт, в параметре запуска которого будет передаваться один из определенных выше PIDов. Скрипт должен в автоматическом режиме обеспечивать, чтобы процесс с переданным PIDом использовал процессорное время в диапазоне от 10% до 20%. Запустив в другой консоли top удостоверьтесь, что результат достигнут.

iv) Создайте пару скриптов: генератор и обработчик. Процесс «Генератор» считывает с консоли строки в бесконечном цикле и передает строки процессу «Обработчик» с помощью именованного канала. Процесс «Обработчик» должен осуществлять следующую обработку переданных строк: если строка содержит единственный символ «**+**», то процесс обработчик переключает режим на «сложение» и ждет ввода численных данных. Если строка содержит единственный символ «*****», то обработчик переключает режим на «умножение» и ждет ввода численных данных. Если строка содержит целое число, то обработчик осуществляет текущую активную операцию (выбранный режим) над текущим значением вычисляемой переменной и считанным значением (например, складывает или перемножает результат предыдущего вычисления со считанным числом) и выводит результат на экран. При запуске скрипта режим устанавливается в «сложение», а вычисляемая переменная приравнивается к 1. В случае получения строки **`QUIT`** скрипт «Обработчик» выдает сообщение о плановой остановке и оба скрипта завершают работу. В случае получения любых других значений строки оба скрипта завершают работу с сообщением об ошибке входных данных.

v) Создайте пару скриптов: генератор и обработчик. Процесс «Генератор» считывает с консоли строки в бесконечном цикле. Если считанная строка содержит единственный символ «**+**», он посылает процессу «Обработчик» сигнал **`USR1`**. Если строка содержит единственный символ «*****», генератор посылает обработчику сигнал **`USR2`**. Если строка содержит слово **`TERM`**, генератор посылает обработчику сигнал **`SIGTERM`** и завершает свою работу. Другие значения входных строк игнорируются. Обработчик добавляет 2 или умножает на 2 текущее значение обрабатываемого числа (начальное значение принять на единицу) в зависимости от полученного пользовательского сигнала и выводит результат на экран. Вычисление и вывод производятся один раз в секунду. Получив сигнал **`SIGTERM`**, «Обработчик» завершает свою работу, выведя сообщения о завершении работы по сигналу от другого процесса.

Часть 3. Дополнительный набор заданий

Организация управления памятью в ОС Linux

В Linux используется страничная организация виртуальной памяти. Память разбита на страницы. Размер страницы можно посмотреть в параметрах конфигурации с помощью команды **`getconf PAGE_SIZE`**. При обращении к адресу в памяти происходит динамическое преобразование адреса путем замены старших бит виртуального адреса на номер физической страницы с сохранением значения младших бит как смещения на странице.

Обычно, кроме физической памяти используется также раздел подкачки. В этом случае адресное пространство процесса состоит из страниц, находящихся в оперативной памяти и страниц, находящихся в разделе подкачки. Параметры раздела подкачки можно узнать из файла **`/proc/swaps`**. При динамическом выделении памяти процессу, операционная система сначала пытается выделить страницы в физической памяти, но если это невозможно, инициирует страничный обмен, в рамках которого ряд страниц из физической памяти вытесняется на раздел подкачки, а адреса, соответствующие вытесненным страницам выделяются процессу под новые страницы.

Операционная система контролирует выделение памяти процессам. Если процесс попробует запросить расширение адресного пространства, которое невозможно в пределах имеющейся свободной оперативной памяти, его работа будет аварийно остановлена с записью в системном журнале.

Основные источники данных о состоянии памяти вычислительного узла

`команда free`

файл **`/proc/meminfo`** (документация в соответствующем разделе **`man proc`**)

файл **`/proc/[PID]/statm`** (документация в соответствующем разделе **`man proc`**)

утилита **`top`**

Проведите два виртуальных эксперимента в соответствии с требованиями и проанализируйте их результаты. В указаниях ниже описано, какие данные необходимо фиксировать в процессе проведения экспериментов.

Зафиксируйте в отчете данные о текущей конфигурации операционной системы и вычислительного узла:

- Количество ядер и семейство процессора.
- Общий объем оперативной памяти.
- Объем раздела подкачки.
- Размер страницы виртуальной памяти.
- Объем свободной физической памяти в ненагруженной системе.
- Объем свободного пространства в разделе подкачки в ненагруженной системе.

Эксперимент №1: Анализ работы подсистемы управления памятью (завершенный эксперимент и полный отчет, включая подробные выводы = 5 баллов)

Создайте скрипт **`mem.bash`**, реализующий следующий сценарий. Скрипт выполняет бесконечный цикл. Перед началом выполнения цикла создается пустой массив и счетчик шагов, инициализированный нулем. На каждом шаге цикла в конец массива добавляется последовательность из 10 элементов, например, (1 2 3 4 5 6 7 8 9 10). Каждый 100000-ый шаг в файл **`report.log`** добавляется строка с текущим значением размера массива (перед запуском скрипта, файл обнуляется).

Создайте копию созданного скрипта в файле **`mem2.bash`** и настройте её на запись в файл **`report2.log`**.

Создайте скрипт, который запустит немедленно друг за другом оба скрипта в фоновом режиме, затем запустит слежение за результатами утилиты **`top`**. Слежение должно фиксировать изменения во времени следующих значений (шаг фиксации подберите самостоятельно, но рекомендуется, чтобы до аварийной остановки первого процесса было зафиксировано не менее 100 наборов значений):

- значения параметров памяти системы (верхние две строки над основной таблицей);
- значения параметров в строке таблицы, соответствующей каждому из двух скриптов;
- изменения в верхних пяти процессах (как меняется состав и позиции этих процессов).

Фиксация должна происходить до аварийной остановки последнего из двух скриптов и их исчезновения из перечня процессов в **`top`**.

Посмотрите с помощью команды **`dmesg | grep "mem[2]*.bash"`** последние записи о скриптах в системном журнале и зафиксируйте их в отчете. Также зафиксируйте значения в последних строках файлов **`report.log`** и **`report2.log`**.

Обработка результатов:

Постройте графики изменения каждой из величин, за которыми производилось наблюдение. Рекомендуется сгруппировать по несколько графиков на одной плоскости. Объясните динамику изменения измеренных величин

исходя из теоретических основ управления памятью в рамках страничной организации памяти с разделом подкачки. Объясните значения пороговых величин: размер массива, при котором произошла аварийная остановка процесса, параметры, зафиксированные в момент аварийной остановки системным журналом. Сформулируйте письменные выводы.

Эксперимент №2: Анализ работы планировщика процессов (завершенный эксперимент и полный отчет, включая подробные выводы = 5 баллов)

Найдите или придумайте вычислительно сложный алгоритм. Алгоритм должен минимально требовать входные данные и выгружать результаты (в идеале, получает на вход несколько числовых параметров и на выходе формирует одно или несколько числовых значений), но требовать для завершения работы значительного времени, порядка 2-3 секунд или более. Например, алгоритм может вычислять с высокой точностью значение некоторой сложной функции в заданной точке. Реализацию алгоритма можно сделать как на любом языке высокого уровня, например, C/C++, так и прямо на bash.

Идея эксперимента. Необходимо получить N результатов выполнения созданного алгоритма с разными комбинациями входных параметров (например, результаты вычисления значения функции в N точках). Важно, что исполнение алгоритма с каждой комбинацией входных параметров должно требовать приблизительно равного времени на выполнение. Эксперимент будет проходить в четыре этапа. На нечетных этапах вычисления каждого из N результатов будут осуществляться последовательно друг за другом, а на четных – параллельно. При этом будет замеряться суммарное время, затрачиваемое на получение всех N результатов. Также необходимо оценить различия в результатах в системе с одним (1 и 2 этап) или с несколькими (2 и 4 этап) процессорами.

В результате можно предложить такой план проведения экспериментов (его можно корректировать, желательно, обсуждая с преподавателем).

1. Первый этап.
    a. В виртуальной машине настроить использование одного процессора.
    b. Написать скрипт, который будет, получив параметр N, запускать последовательно друг за другом N вычислений для разных значений входных параметров. Запуск вычисления для следующего параметра должен происходить сразу после завершения предыдущего вычисления.
    c. С помощью другого скрипта для каждого N в диапазоне от 1 до 10 запускать 5 раз запускающий скрипт из пункта 2 через утилиту time и фиксировать время, затраченное на полное выполнение запускающего скрипта. На выходе должно получиться 10 серий по 5 значений.
    d. В каждой серии посчитать среднее арифметическое значений и построить график зависимости этих усредненных оценок от N.
2. Второй этап.
    a. Написать скрипт, который, получив параметр N, параллельно (не дожидаясь завершения предыдущего) запустит N вычислений для разных входных параметров.
    b. Повторить пункты “c” и “d” из первого этапа с использованием написанного в предыдущем пункте запускающего скрипта.
3. Третий и четвертый этапы – повторить первый и второй этапы, выделив в настройках виртуальной машины 2 процессора.

В результате выполнения этой группы экспериментов должны получиться 4 графика, которые рекомендуется разместить на одной плоскости.

Можно и приветствуется, если Вы дополните эксперимент так, чтобы кроме оценки времени выполнения наборов заданий, оценивались и другие параметры ОС: память, потребляемые отдельными процессами ресурсы и т.д. Выбор таких параметров и методов их оценки на усмотрение экспериментаторов.

Обработка результатов:

После получения всех результатов необходимо оформить отчет, в котором описать:

1. Детали реализации алгоритма.
2. Параметры виртуальной машины и хостового компьютера, на которых проводился эксперимент.
3. Уточненный (возможно) план эксперимента.
4. Графики с описанием, в рамках какого этапа каждый из них получен.
5. Выводы. Попытаться теоретически объяснить читаемые на графиках явления.