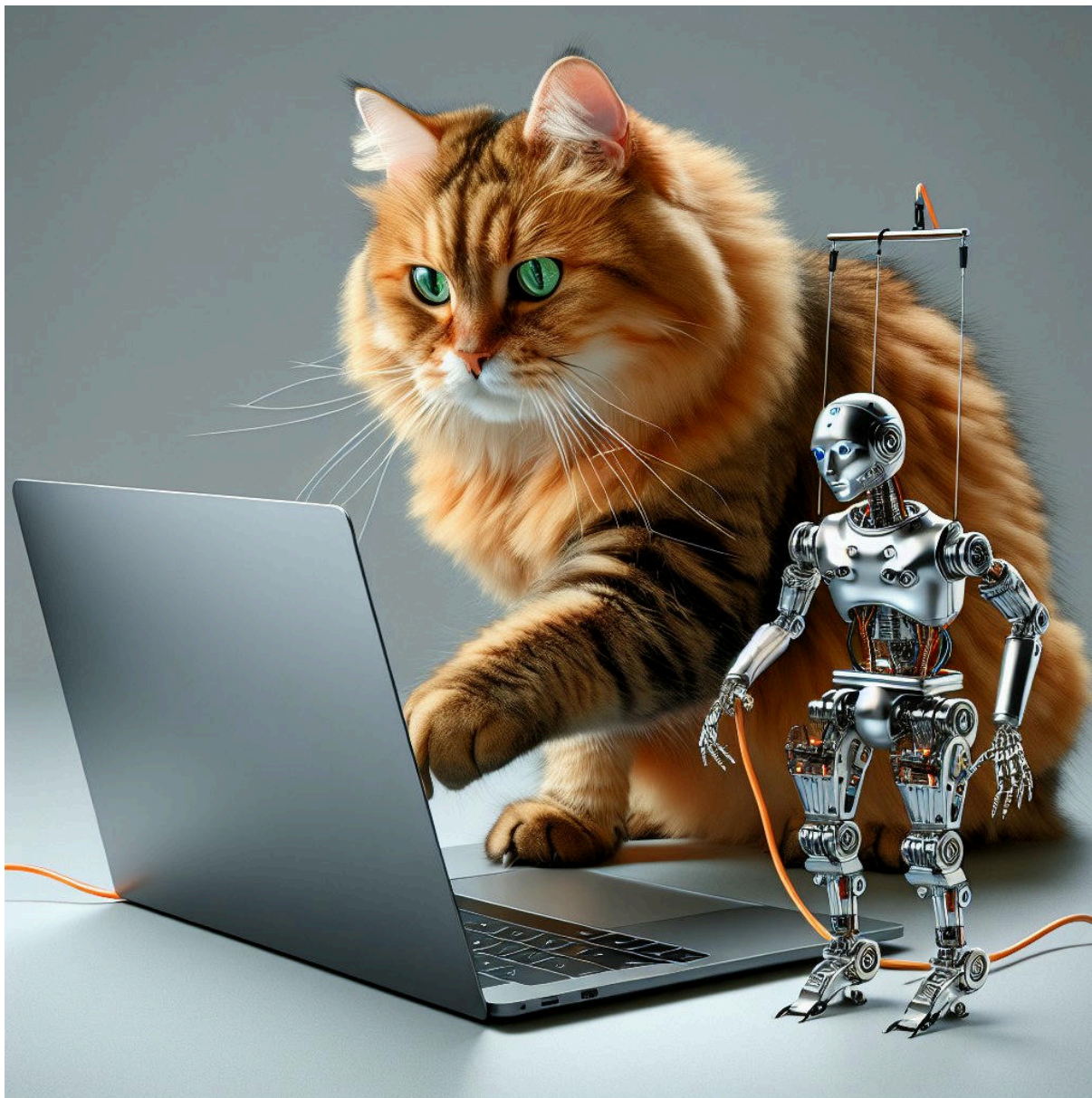


# Лабораторная работа III

Управление процессами в Linux



## Методические материалы к заданию

Подробное описание возможностей и синтаксиса команд можно ознакомиться в документации, доступной по команде `man <команда>`.

## Команды для управления процессами

Здесь представлены основные команды по управлению процессами на уровне пользовательского интерфейса:

- **kill** – передает сигнал процессу. Сигнал может передаваться в виде его номера или символического 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`;
- **at** – осуществляет однократный отсроченный запуск команды.
- **cron** – демон, который занимается планированием и выполнением команд, запускаемых по определенным датам и в определенное время. Команды, выполняемые периодически, указываются с использованием команды `crontab`. Команды, которые должны быть запущены лишь однажды, добавляются при помощи `at`. Синтаксис строки в `crontab` подробно описан здесь: <http://www.opennet.ru/man.shtml?topic=crontab&category=5&russian=2>.
- **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`.

Код генератора:

```
#!/bin/bash

LINE=""

while true; do
    read LINE
    echo $LINE > pipe
done
```

Код обработчика:

```
#!/bin/bash

LINE=""

(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 исключением 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.

Код генератора:

```
#!/bin/bash

LINE=""

while true; do
    read LINE
    case $LINE in
        STOP)
            kill -USR1 $(cat .pid)
            ;;
        *)
            :
            ;;
    esac
done
```

Код обработчика:

```
#!/bin/bash

echo $$ > .pid

A=1
MODE="work"

usr1()
{
    MODE="stop"
}

trap 'usr1' USR1

while true; do
    case $MODE in
        "work")
            let A=$A+1
            echo $A
            ;;
        "stop")
            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.

## Условия заданий

В рамках данной лабораторной работы можно предоставить реализацию всех скриптов как на Bash, так и на языке Си. *При выборе второго варианта вы можете претендовать на 150% от максимальной стоимости задания.*

Напишите скрипты, решающие следующие задачи:

1. Создайте скрипт, который создаст директорию `test` в домашней директории (если ее не было раньше), затем создаст в `test` файл с именем **Дата\_Время\_Запуска\_Скрипта** и выведет в файл `~/report` сообщение вида "`<Дата:время> test was created successfully`". Также скрипт должен проверять наличие похожих файлов, но с предыдущей датой: если такие файлы нашлись, их нужно сжать в архив с именем **Дата** и поместить в папку `test/archived`. *Дата* выводится в формате: `<Год>—<Месяц>—<Число>`.

2. Задайте еще один однократный запуск скрипта из пункта 1 через 2 минуты. Консоль после этого должна оставаться *свободной*. Выполнив отдельную команду организуйте слежение за файлом `~/report` и выведите на консоль новые строки из этого файла, как только они появятся.
  3. Задайте запуск скрипта из пункта 1 в каждую пятую минуту каждого часа в день недели, в который вы будете сдавать работу.
  4. Создайте три фоновых процесса, выполняющих одинаковый бесконечный цикл вычисления (например, перемножение двух чисел). После запуска процессов должна сохраниться возможность использовать виртуальную консоль, с которой их запустили. Используя команду `top`, проанализируйте процент использования ресурсов процессора этими процессами. Создайте скрипт, который будет в автоматическом режиме обеспечивать, чтобы тот процесс, который был запущен первым, использовал ресурс процессора не более чем на 10%. Послав сигнал, завершите работу процесса, запущенного третьим. Проверьте, что созданный скрипт по-прежнему удерживает потребление ресурсов процессора первым процессом в заданном диапазоне.
- 

В следующих заданиях требуется создать пару скриптов: “генератор” и “обработчик”.

Важные условия:

- к заданию также нужно написать “лаунчер”, который запускает генератор и обработчик сразу;
  - генератор и обработчик должны работать в одной консоли, значит, не должно быть путаницы между вводом пользователя и выводом программы.
- 

5. Создайте пару скриптов: генератор и обработчик. Процесс «Генератор» передает информацию 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). 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.
6. 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.

## Правила оформления и написания программ на С

Программы должны удовлетворять следующим требованиям:

- В случае, если это требуется по ТЗ, корректно обрабатывать ошибки при взаимодействии с внешним миром: ошибки ввода-вывода, некорректный пользовательский ввод и прочее. Если это произошло – необходимо вывести сообщение об ошибке (на английском языке) и завершить исполнение с ненулевым кодом возврата.
- Программа никогда не должна падать. Падение – признак ошибок в реализации.
- Программа не должна содержать лишних сущностей: закомментированных больших участков кода, неиспользуемых переменных и функций и тому подобное. Это засоряет код и увеличивает время проверки.

При успешном выполнении программа возвращает код 0. Если же что-то пошло не так, то она сообщает о проблеме через ненулевой код возврата и сообщение об ошибке в поток вывода ошибок `stderr`.

### За что можно потерять баллы к критерии качества кода

Кроме правильности результата будет учитываться скорость работы программы. То есть, если проверяющий не дождался за разумное время завершения работы программы, то тест будет считаться не пройденным.

В программе можно использовать стандартные библиотеки и заголовочные файлы, а также те, которые относятся к POSIX/GNU C библиотеке. Например, `<bits/stdc++.h>` таковым не является и его использование влечёт за собой потерю баллов.

Если программа использует функции, которые явно не объявлены в файле с исходным кодом (например, тип `size_t` без подключения `<stdlib.h>` и пр.), то за это также будут снижаться баллы (даже если у вас всё работает).

**Обратите внимание:** обёртка Bash на Си — это не самурайский путь, вам необходимо использовать GNU C библиотеку при выполнении лабораторной работы. Все сдачи начиная с этого момента на Си с обёрткой Bash *не будут засчитываться* никаким образом.