Лабораторная работа №4
«Процессы в операционной системе Astra Linux»

1. Теоретическая часть

1.1. Историческая справка

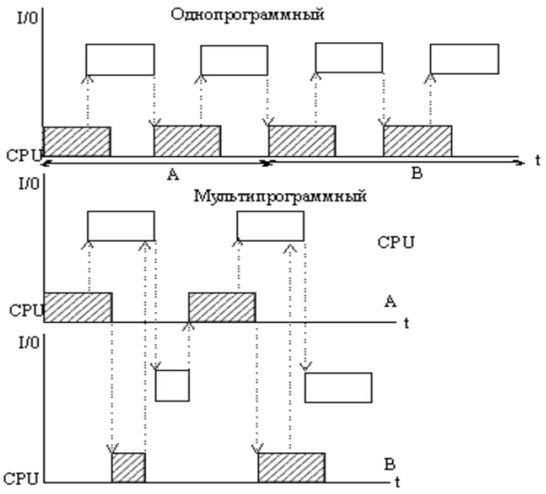
На заре вычислительной техники первые компьютеры (мейнфреймы) могли обрабатывать только одну программу, введённую с помощью различных носителей информации (перфокарты, магнитные ленты и т.д.). Инструкции и данные программы считывались в оперативную память компьютера и оттуда обрабатывались процессором. Собственно говоря, действие по производству вычислений согласно определённой программе называется **процессом**. Однако в те времена концепция процессов ещё не существовала.

Из-за того, что процессор полностью занят одной программой, другим программистам приходилось ждать неделями, чтобы они могли загрузить свои программы и получить спустя несколько часов свои результаты. Об исправлении ошибок «на лету» не приходилось говорить. При этом процессор занят только арифметическими и логическими операциями. Во время операций ввода и вывода он простаивает. Поэтому были сделаны усилия по оптимизации (точнее, по минимизировании) времени простоя процессора.

Сначала были созданы **системы пакетной обработки**. Программы собирались в пакеты и одним массивом загружались в компьютер, который поочерёдно обрабатывал каждую инструкцию. То есть вместо смены одной программы на другую человеком (на что уходило немало времени) появилась смена одного пакета на другой, что было быстрее.

Однако проблема простоя процессора во время операций ввода и вывода не исчезла. Дело в том, что в пакете может быть программа, которая может потребовать введения каких-либо данных. А так как инструкции выполняются

последовательно, то и им приходилось ждать обработки. Тогда появилась идея **мультипрограммирования**.



Она состоит в том, что в память компьютера пакетом загружались программы, при этом процессор мог переключаться с одного процесса на другой (так как они в одной памяти находятся) с сохранением контекста (содержимого регистров, указателя на следующую инструкцию и т.д. Переключение происходило тогда, когда начиналась операция ввода/вывода. Пока один процесс ожидает ввода, процессор обрабатывает другую программу.

Однако нужна была программа (супервизор), которая бы устанавливала порядок вычислений и следила бы за переключением контекста. В то время напрямую программировать приоритеты было трудно. Также оставалась проблема слабого взаимодействия пользователя и компьютера. Ему всё равно так же приходилось загружать пакеты программ и следить за работой. Часто бывало, что пакет содержал программы, которые содержали в себе очень много операций вычислений. То есть другие процессы могли просто не дождаться того, пока им не выделят процессорное время.

Так возникла идея **системы с разделением времени**. В её основе лежало мультипрограммирование, но процессам уделялся определённый квант процессорного времени. Таким образом процессы не зависели друг от друга, им уделялось одинаковое внимание со стороны процессора.

1.2. Концепция процессов в Linux

Процесс — это запущенный экземпляр программы, находящийся в процессе ее выполнения с выделенными ему ресурсами в виде открытых файлов, состояния процессора, адресного пространства памяти и потоков.

Процессом управляет ядро операционной системы, точнее **планировщик задач**. Он отвечает за отслеживание и выделение запускаемым процессам определенного приоритета, чтобы процессы «не мешали» друг-другу работать, а так же за распределение пространства памяти, чтобы пространство памяти одного процесса не пересекалось с пространством другого.

Для каждой запускаемой программы создаётся отдельный процесс в соответствии со структурой данных, называемой **процессорным дескриптором**. Её можно изучить в исходном коде ядра Linux в директории **linux/include/linux/sched.h** под названием **task_struct**. Например, в структуре приведены следующие поля, которые будут интересны в рамках текущей лабораторной работы:

Поле	Описание
state	Описывает то, что в данный момент происходит с процессом, который может быть в одном из следующих состояний: • **Активен (R=Running)** — процесс либо выполняется в данный момент, либо ожидает выделения ему очередного кванта времени центрального процессора. • **Спит (S=Sleeping)** — процесс находится в состоянии прерываемого ожидания, то есть ожидает какого-то события, сигнала или освобождения нужного ресурса. • **Приостановлен (T=Stopped)** — процесс остановлен. • **Зомби (Z=Zombie)** — процесс освобождает свои ресурсы, но не свой PID в таблице процессов.
pid	Идентификатор процесса **Proccess ID** или просто **PID**

В файловой системе Linux существует папка **/proc**, которая хранит в себе информацию о каждом запущенном процессе. На самом деле её не существует на диске или даже в оперативной памяти. Все поддиректории, файлы и хранящаяся в них информация генерируется ядром на лету, как только вы её запрашиваете.

Например, мы можем узнать актуальное состояние процесса **bash** (командной оболочки). Для этого нам надо знать его **PID**, которая хранится в специальной переменной **$**. Его содержимое можно прочитать командой

```
echo $$
```

Теперь мы можем перейти в папку **/proc/$$**. Там содержатся файлы, описания некоторых из которых приведены ниже:

Системный вызов	**Описание**
cmdline	Содержит команду с помощью которой был запущен процесс, а также переданные ей параметры
cwd	Символическая ссылка на текущую рабочую директорию процесса
exe	Ссылка на исполняемый файл
root	Ссылка на папку суперпользователя
environ	Переменные окружения, доступные для процесса
fd	Содержит файловые дескрипторы, файлы и устройства, которые использует процесс
stat, status	Состояние процесса

Нас интересует файл **status**, в котором содержится множество технической информации. Поэтому отфильтруем текст следующей командой:

```
cat /proc/$$/status | grep Stat
State:  S (sleeping)
```

Процесс **bash** находится в ожидании команд. Как только он их получит, сразу же перейдёт в активное состояние.

1.3. Мониторинг работы процессов

ps (от англ. process status) — утилита, с помощью которой можно посмотреть информацию о процессах (PID, загрузка процессора и памяти и т.д.).

Чтобы просто посмотреть процессы в текущей оболочке, используется команда

```
ps
```

А чтобы вывести информацию о всех запущенных программах, используется ключ

```
ps -aux
```

Например, PID запущенной программы **bash** также можно узнать с помощью команды

```
ps -aux | grep bash
```

Можно самим указывать столбцы для вывода:

```
ps –xao command, ppid, pid
```

top (table of processes) — консольная команда, которая выводит список работающих в системе процессов и информацию о них.

В отличие от утилиты **ps** позволяет выводить информацию о системе, а также список процессов динамически обновляя информацию о потребляемых ими ресурсах.

Для вывода информации достаточно команды

```
top
```

Выведенное окно можно условно разделить на две части. В верхней части находится информация о системе, общем использовании ресурсов процессора и памяти, раздела подкачки, и так далее. В нижней части окна расположен

список запущенных процессов с информацией, отсортированных по определённому полю.

htop (Hisham's top) — альтернатива утилиты top, также позволяющая проанализировать работу систему не только с помощью горячих клавиш, но и указателя мышки.

Устанавливается командой

```
sudo apt install htop
```

и запускается командой

```
htop
```

1.4. Приоритет процессов

В ОС выполняется множество процессов, при этом очень важным из них (например, системным) выделяется больше процессорного времени за счёт высокого приоритета.

Пользователь также может назначать приоритет своим процессам с помощью утилиты ***nice (от англ. «вежливый»)***. По этимологии этой команды процесс с большим значением **nice** — более вежлив к другим процессам, позволяя им использовать больше процессорного времени, поскольку он сам имеет меньший приоритет (и, следовательно, большее «значение вежливости» — niceness value). То есть, процесс с приоритетом 15 будет более приоритетным, чем процесс с приоритетом 20. Значение параметра выбирается из диапазона от -20 до 20.

Приоритет **nice** процесса используется планировщиком процессов ядра ОС при распределении процессорного времени между процессами. Его можно увидеть с помощью утилит **top** или **htop** (столбец **NI**).

Чтобы запустить команду с изменённым значением **nice**, используется следующий синтаксис:

```
nice -n <приоритет> <выполняемая команда>
```

Если нужно изменить приоритет уже запущенного процесса, можно воспользоваться командой **renice**:

```
renice -n <приоритет> <PID процесса>
```

1.5. Файловые дескрипторы

У каждого процесса в папке **/proc/PID** находится папка **fd**, которая хранит файловые дескрипторы. Для **bash** наглядно это можно увидеть командой

```
ls -hal /proc/$$/fd
```

Как минимум, там должны быть файловые дескрипторы (номера файлов) **0**, **1** и **2**. Это известные нам стандартные потоки ввода и вывода.

Файловый дескриптор — натуральное число (идентификатор), закреплённое за определённым потоком ввода-вывода.

При создании нового потока ввода-вывода (который может быть связан как с файлами, так и с каталогами и FIFO), ядро возвращает его файловый дескриптор создавшему его процессу.

Например, если открыть новую вкладку эмулятора терминала, то фактически создаётся новый экземпляр программы **bash** (в этом можно убедиться по новому PID). Во второй вкладке мы можем обратиться к стандартному потоку ввода первой вкладки по его PID:

```
echo “Тест” > /proc/PID/fd/0
```

Тогда в первой вкладке появится сообщение из второй вкладки терминала.

Теперь проверим, что ядро действительно присваивает номер новому потоку ввода/вывода.

Запустим следующий процесс в фоновом режиме с помощью символа **&**

```
cat /dev/zero &
```

Эта команда будет бесконечно выводить нули в терминал из специального файла. Посмотрим содержимое папки **fd** по PID процесса:

```
lr-x------ 1 student student 64 окт 27 16:44 3 -> /dev/zero
```

Действительно, ядро ОС присвоило номер 3 потоку вывода файла **/dev/zero**. По факту, файловые дескрипторы есть символические ссылки на файлы.

1.6. Системные вызовы

Системный вызов (англ. system call) — обращение прикладной программы к ядру операционной системы для выполнения какой-либо операции.

В ядре Linux используется большое множество системных вызовов. Вот основные из них:

Системный вызов	Описание
fork	Создание нового дочернего процесса
read	Попытка читать из файлового дескриптора
write	Попытка записи в файловый дескриптор
open	Открыть файл для чтения или записи
close	Закрыть файл после чтения или записи
chdir	Изменить текущую директорию
execve	Выполнить исполняемый файл
stat	Получить информацию о файле

За процессом работы любой из программ можно проследить, наблюдая системные вызовы, которые использует программа. Утилита **strace** позволяет это сделать.

Например, проследим за исполнением команды

```
echo 1
```

Это делается командой

```
strace echo 1
```

которая выведет очень много информации. Синтаксис вывода такой:

имя_системного_вызова (параметр1, параметр2) = результат сообщение

Посмотрим, использует ли процесс вызов **work**:

```
strace -e trace=write echo 1
```

Нам выведется следующая строка

```
write(1, “1\n”, 21) = 2
```

Первый параметр вызова **write** отвечает за то, куда надо записать строку (в какой поток по файловому дескриптору), переданную в качестве второго параметра.

1.7. Сигналы

Сигнал — асинхронное уведомление процесса о каком-либо событии, один из основных способов взаимодействия между процессами.

Существует стандартные сигналы в ядре Linux:

Сигнал	Описание
SIGINT (2)	Клавиатурный сигнал, срабатывает когда мы нажимаем **Ctrl+c**. Это штатное завершение, то есть процесс будет завершён корректно (если процесс вообще умеет завершаться корректно).
SIGQUIT (3)	Клавиатурный сигнал, срабатывает когда мы нажимаем **Ctrl+**. Аварийное завершение с выдачей отладочной информации.

SIGKILL (9)	Этот сигнал сразу завершает процесс (некорректно). И это поведение нельзя изменить, то есть программист не может сам указать программе что делать в случае получения этого сигнала.
SIGTERM (15)	Это аналог SIGINT (2). Если у процесса нет управляющего терминала, то отправить ему клавиатурный сигнал не получится, поэтому используется этот сигнал.
SIGCHLD (17)	Сигнал, посылаемый при изменении статуса дочернего процесса (завершён, приостановлен или возобновлен).
SIGCONT (18)	Сигнал для возобновления работы процесса.
SIGTSTP (20)	Клавиатурный сигнал, когда мы нажимаем **Ctrl+z**. Этот сигнал приостанавливает процесс на управляющем терминале и переводит процесс на задний фон. То есть процесс переходит в состояние T (stopped by job control signal — остановленный специальным сигналом).

Существует специальная утилита **kill**, которая позволяет отправлять сигналы процессам. Тип сигнала указывается в качестве параметра в виде номера (например, **-9**) или имени (например, **-SIGKILL**). Следующим параметром нужно указать PID процесса, которому мы отправляем сигнал.

Примеры:

```
kill -n 9 9898
kill -s SIGKILL 2565
```

Управление процессами в Linux можно применять и для того, чтобы в терминале можно было работать также, на заднем и переднем фоне. Представьте, что вы читаете справку **man** и захотели что-то попробовать.

Чтобы не закрывать **man**, вы можете нажать **Ctrl+z** и приостановить **man**. При этом работа **man** не просто приостанавливается, она уходит на задний фон.

Дальше вы можете поработать в терминале. А затем, когда захотите, вернёте **man** на передний фон. А чтобы увидеть процессы на заднем фоне используется команда **jobs**. Чтобы вернуть задание на передний фон, нужно выполнить команду **fg**.

1.8. Запуск процессов по расписанию

cron — утилита, использующаяся для периодического выполнения заданий в определённое время.

Можно посмотреть, есть ли задачи для текущего пользователя:

```
crontab -u $USER -l
```

Все существующие задачи удаляются ключом **-r**.

Для настройки времени, даты и интервала, когда нужно выполнять задание, используется команда

```
crontab -u $USER -e
```

которая открывает в текстовом редакторе конфигурационный файл.

Там прописываются задачи в следующем виде:

```
минута час день месяц день_недели /путь/к/исполняемому/файлу
```

```
* * * * * выполняемая команда
- - - - -
| | | | |
| | | | ----- день недели (0–7) (воскресенье = 0 или 7)
| | | ------- месяц (1–12)
| | --------- день месяца (1–31)
| ----------- час (0–23)
------------- минута (0–59)
```

Дата и время указываются с помощью цифр или специальных символов.

Рассмотрим следующие примеры:

- Запуск программы каждую минуту:

```
* * * * * ls
```

- Запуск в 15.30 во вторник:

```
30 15 * * 2 ps
```

- Строка ниже будет выводить «hello world» в файл каждую 5-ю минуту каждого первого, второго и третьего часа (то есть 01:00, 01:05, 01:10, вплоть до 03:55).

```
*/5 1,2,3 * * * echo hello world > /home/USER/file.txt
```

2. Практическая часть

2.1. Задание 1

2.1.1. Откройте эмулятор терминала и выполните мониторинг системы с помощью утилит **ps**, **top** и **htop**.

2.1.2. Выясните, что делает утилита **pstree**. Что является родительским процессом процесса **bash**?

2.1.3. С помощью утилиты **ps** выведете информацию о процессе **bash** и его родителе. Сравните их **PID** и **PPID.**

2.2. Задание 2

2.2.1. Откройте новую вкладку терминала и отправьте сообщение «Привет <номер процесса>! Мой PID: $$» в стандартный поток ввода процесса с номером процесса **bash** первой вкладки. Проверьте первую вкладку терминала.

2.2.2. Запустите процесс выполнения команды

```
cat /dev/zero | sleep 100000 &
```

и узнайте PID процессов **cat** и **sleep**.

2.2.3. Выясните, какие файловые дескрипторы присвоены данным процессам. Каким образом работают каналы между процессами?

2.3. Задание 3

2.3.1. Запустите процесс выполнения команды

```
cat /dev/zero
```

2.3.2. Откройте новую вкладку терминала, подключитесь к процессу **cat** по его PID и по системным вызовам определите, что процесс в текущий момент делает.

2.3.3. В первой вкладке терминала с помощью клавиш *остановите* процесс **cat**. С помощью **strace** узнайте, какой сигнал был послан процессу.

2.3.4. Возобновите работу процесса и откройте третью вкладку терминала.

2.3.5. В третьей вкладке введите команду для посылки сигнала прерывания процессу **cat** и отследите сигнал во второй вкладке в **strace**.

2.3.6. В первой вкладке с помощью утилиты **kill** завершите работу второй и третьей вкладок терминала.

2.4. Задание 4

2.4.1. Сделайте так, чтобы в полноэкранном режиме экран был разделён двумя вкладками терминала. Во второй вкладке запустите **top**.

2.4.2. Запустите процесс выполнения команды

```
cat /dev/zero
```

и оцените его нагрузку на процессор.

2.4.3. Задайте меньший приоритет этому процессу с целью уменьшения нагрузки.

2.5. Задание 5

2.5.1. Запланируйте задачу так, чтобы за 10 минут до окончания лабораторной работы выводилось сообщение «Пора сдаваться!». При этом сообщение должно отправляться в устройство **/dev/pts/<номер терминала>**.

Контрольные вопросы

1. Что такое процесс?
2. Какие существуют состояния процесса?
3. С помощью чего программы взаимодействуют с ядром ОС?
4. Для чего нужны сигналы?
5. В каких случаях может понадобиться управление приоритетами?