

5407

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

РЯЗАНСКИЙ ГОСУДАРСТВЕННЫЙ РАДИОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
им. В.Ф. УТКИНА

ИЗУЧЕНИЕ ПРОЦЕССОВ В LINUX

Методические указания
к лабораторным работам

Рязань 2021

УДК 004.45

Изучение процессов LINUX: методические указания к лабораторным работам / Рязан. гос. радиотехн. ун-т им. В.Ф. Уткина; сост.: С.А. Бубнов, А.А. Бубнов. Рязань, 2021. 28 с.

Содержат лабораторные работы по дисциплине «Операционные системы».

Предназначены для студентов очной и заочной форм обучения, изучающих курс «Операционные системы».

Ил. 19. Библиогр.: 5 назв.

Операционная система LINUX, процесс, адресное пространство, планирование процессов, поток, многопоточные процессы

Печатается по решению редакционно-издательского совета Рязанского государственного радиотехнического университета.

Рецензент: кафедра ВПМ РГРТУ (зав. кафедрой проф. Г.В. Овечкин)

СОДЕРЖАНИЕ

ЛАБОРАТОРНАЯ РАБОТА № 1.....	2
ЛАБОРАТОРНАЯ РАБОТА № 2.....	14
ЛАБОРАТОРНАЯ РАБОТА № 3.....	22
БИБЛИОГРАФИЧЕСКИЙ СПИСОК.....	28

ЛАБОРАТОРНАЯ РАБОТА № 1

ЗНАКОМСТВО С ПРОЦЕССАМИ В LINUX

1. Понятие процесса. Единого определения понятия «процесс» не существует. С одной стороны, процесс можно определить как выполняющуюся программу. С другой, более общей, процесс — это контейнер, который содержит всю необходимую для выполнения программы информацию. У каждого пользователя в *Linux*, как правило, запущено несколько независимых процессов.

При загрузке операционной системы *Linux* первым начинает выполняться процесс *systemd* (либо процесс *init*). Все остальные процессы являются дочерними по отношению к нему, образуя при этом древовидную структуру. Каждый процесс имеет родителя, т.е. процесса, который его породил. С каждым процессом связано его *адресное пространство* — список адресов ячеек памяти, в которые процесс может записывать данные и откуда может их считывать.

Адресное пространство процесса делится на три сегмента (рис.1):

- *текстовый сегмент* (код программы);
- *сегмент данных* (данные);
- *сегмент стека*.

Между сегментом данных и стеком имеется свободный участок памяти, который может потребоваться в случае запроса процесса на увеличение предоставляемого ему объема памяти.



Рис.1. Сегменты адресного пространства процесса

Каждый процесс имеет свой *дескриптор* — это структура данных, полностью описывающая процесс и его состояния. В операционной системе *Linux* дескрипторы процессов хранятся в каталоге */proc*, представляющим собой файловую систему, хранящуюся в оперативной памяти.

Дескриптор процесса содержит:

- уникальный идентификатор процесса — *PID (Process*

Identifier), по которым операционная система отличает один процесс от другого;

- идентификатор родителя *PPID (Parent Process Identifier)*;
- идентификаторы владельца процесса: *реальный идентификатор RUID* и *эффективный идентификатор EUID*;
- идентификаторы группы процесса: *реальный идентификатор RGID (Real Group ID)* и *эффективный идентификатор EGID (Effective Group ID)*;
- параметры планирования;
- текущее состояние процесса.

Реальные идентификаторы наследуются процессом от запустившего его пользователя. *Эффективные идентификаторы* — это новые идентификаторы, полученные процессом во время выполнения. У большинства процессов *RUID* и *EUID* будут одинаковыми. Исключение составляют процессы, у которых установлен бит смены идентификатора пользователя. Идентификатор *EGID* связан с *RGID* также, как *EUID* с *RUID*. Данные идентификаторы необходимы для организации прав доступа процессов к объектам файловой системы.

В общем случае процесс в *Linux* может находиться в одном из состояний:

- спит (*S — sleeping*) процесс находится в состоянии прерываемого ожидания.
- выполнение (*R — running*). Процесс выполняется в данный момент либо ожидает квант времени процессора;
- непрерываемое ожидание (*D — direct*). Процесс ожидает сигнала от аппаратного устройства и не реагирует на другие сигналы;
- приостановлен (*T — stopped by job control signal*). Процесс остановлен сигналом или находится в режиме отладки;
- зомби (*Z - zombie*). Завершившийся процесс, который по каким-то причинам еще не удален из таблицы процессов.

Получить информацию о процессах позволяет утилита *ps*.

Команда [-опции]	Действие
<i>ps -f</i>	Вывод информации о процессах, запущенных пользователем
<i>ps -u</i>	Вывод более детальной информации о процессах по сравнению с предыдущей опцией

<i>ps -p <PID></i> <i>ps -f <PID></i>	Вывод информации о процессе с идентификатором <i>PID</i>
<i>ps -ax</i>	Вывод информации о всех процессах в системе
<i>ps jf</i>	Вывод информации о процессах-лидерах-сессий и лидерах-групп
<i>ps -t</i>	Вывод информации о процессах, запущенных с текущего терминала
<i>ps -s</i>	Вывод на экран диспозиции сигналов для процессов пользователя

Пример работы утилиты *ps -u* (рис. 2):

```
main@main-VirtualBox:~$ ps -u
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
main      2027  0.0  0.4   8332   4748 pts/7    Ss   09:02   0:00 bash
main      2145  0.0  0.4   8332   4732 pts/5    Ss+  09:04   0:00 bash
main      4985  0.0  0.3   8980   3180 pts/7    R+   09:21   0:00 ps -u
main@main-VirtualBox:~$
```

Рис. 2. Результат работы утилиты *ps -u*

На экран выводится снимок состояний всех процессов, запущенных пользователем, на момент работы утилиты *ps*.

Поясним значения некоторых полей.

USER отражает имя учетной записи пользователя, от имени которой был запущен данный процесс.

PID содержит уникальный идентификатор процесса.

CPU — процент загрузки центрального процессора процессом.

MEM — часть реальной памяти, используемая процессом.

VSZ — виртуальный размер процесса в килобайтах.

RSS — количество блоков памяти размером 1 Кбайт.

TTY — номер терминала, с которого был запущен процесс (*управляющий терминал*). *Tty1*, *tty2*... — текстовые терминалы, *pts/1*, *pts/2* — экземпляры эмулятора терминала, запущенные в графическом режиме. У многих процессов управляющий терминал отсутствует, поскольку они выполняют приложения, взаимодействующие с пользователем посредством графической подсистемы.

STAT — отображает текущее состояние процесса.

Справа от символа, характеризующего состояние процесса (символы *S*, *R*, *T*, *D*, *Z*) могут стоять дополнительные символы, значения которых пояснены в таблице.

Дополнительный символ	Значение
+	Процесс находится в интерактивном режиме управляющего терминала
<	Высокий приоритет процесса
<i>N</i>	Низкий приоритет процесса
<i>L</i>	Процесс имеет заблокированные в памяти страницы
<i>s</i>	Процесс-лидер сессии
<i>l</i>	Многопоточный процесс

TIME — время центрального процессора, затребованное процессором. Оно увеличивается только при нахождении процесса в состоянии *R*.

START — время запуска процесса.

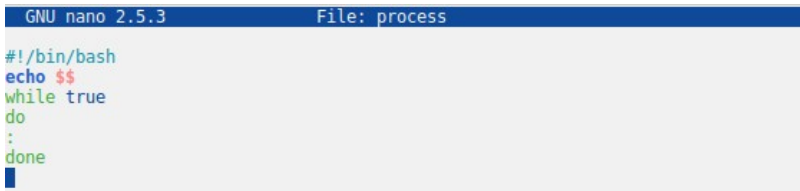
COMMAND — имя и аргументы команды.

Для отображения информации о всех запущенных процессах в системе в режиме реального времени служит утилита *top*. Варианты ее использования приведены в таблице.

Команда [-опции]	Действие
<i>top</i>	Отображение списка всех процессов в системе в режиме реального времени
<i>top -p <PID></i>	Отображение информации о конкретном процессе с идентификатором <i>PID</i> в режиме реального времени
<i>top -u <имя_пользователя></i>	Отображение информации о процессах, запущенных пользователем с именем <i>имя_пользователя</i>

Во время работы утилиты *top* можно настраивать выводимые на экран поля (столбцы), содержащие дополнительную информацию о процессах. Для этого необходимо запустить утилиту *top*, затем нажать клавишу «*f*», осуществить выбор нужного поля (выбор поля осуществляется нажатием клавиш со стрелками с последующим нажатием клавиши «*space*», выбранные поля при этом отмечаются символом «*») и нажать клавишу «*esc*».

2. Запуск и завершение процесса. Для изучения абстракции «процесс» будем использовать процесс пользователя. В качестве такого процесса проще всего взять запущенный пользователем скриптовый файл (исполняемый файл). Чтобы скриптовый файл выполнялся постоянно с момента его запуска, создадим на языке командного интерпретатора *Bash* бесконечный цикл. Для этого перейдем в домашний каталог (*cd ~*), выполним команду *nano process* (либо *vim process*) и запишем строки, показанные на рис. 3.



```
GNU nano 2.5.3      File: process
#!/bin/bash
echo $$
while true
do
:
done
```

Рис. 3. Бесконечный цикл на языке *Bash*

По завершении ввода команд необходимо нажать сочетание клавиш *Ctrl+O* для сохранения файла, а затем *Ctrl+X* для выхода из текстового редактора *nano*. В результате в домашнем каталоге появится скриптовый файл с именем *process*.

Запуск скриптового файла в терминале осуществляется командой: *bash <имя_скриптового_файла>*.

Возможны два варианта запуска исполняемого файла в терминале:

- в активном режиме (команда *bash <имя_скриптового_файла>*). В этом случае оболочка (терминал) перестает реагировать на ввод команд, а управлять таким процессом возможно с помощью сигналов, генерируемых нажатием сочетания клавиш. Сочетания клавиш и соответствующие им сигналы приведены в таблице.

Сочетание клавиш	Сигнал
<i>CTRL+C</i>	Отправка сигнала <i>SIGINT</i> активному процессу (завершение выполнения)
<i>CTRL+Z</i>	Отправка сигнала <i>SIGSTOP</i> (перевод активного процесса в состояние приостановки (<i>T</i>))
<i>CTRL+D</i>	Отправка сигнала, сообщающего о завершении ввода данных и

	завершающего процесс
--	----------------------

- в фоновом режиме (команда `bash <имя_скриптового_файла>&`). В этом случае оболочка готова принимать и выполнять команды, вводимые пользователем.

Существует еще один способ запуска скриптового файла — команда «.» . Отличие от команды `bash` состоит в том, что при использовании «.» скриптовый файл запускается в текущем окружении, а при использовании `bash` — в новом. Чтобы запустить скриптовый файл с помощью «.» необходимо сначала дать ему соответствующее право (команда `chmod u+x <имя_скриптового_файла>`). Запуск скриптового файла `process` в активном режиме различными способами показан на рис. 4.

```
main@main-Ubuntu:~$ bash process
2276
```

а

```
main@main-Ubuntu:~$ ./process
2279
```

б

Рис. 4. Запуск скриптового файла: командой `bash` (а); командой «.» (б)

При этом для выполнения бесконечного цикла операционная система создает и запускает процесс. Число 2276 (2279), отображаемое на экране, это *PID* запущенного процесса. Команда `echo $$` выводит его на экран. При последующих запусках скриптового файла текущего сеанса, а также запусках на других машинах *PID* будет, естественно, другим.

Обычно процессы пользователя, связанные с одним терминалом называются заданиями (*jobs*) и нумеруются, начиная с 1. Порядковый номер задания отображается на экране (число в квадратных скобках) сразу после его запуска в фоновом режиме или перевода уже выполняющегося задания в фоновый режим.

Для управления заданиями, используются утилиты *jobs*, *bg* и *fg*.

Команда [-опции]	Действие
<i>jobs</i>	Вывод на экран списка заданий
<i>fg</i> < <i>n</i> >	Перевод задания с порядковым номером <i>n</i> из фонового режима в активный

<i>bg <n></i>	Перевод задания с порядковым номером <i>n</i> из активного режима в фоновый
---------------------	---

Поскольку активный процесс «блокирует» управляющий терминал, т. е. пользователю невозможно ввести никакие команды, чтобы воспользоваться командой *bg <n>*, необходимо сначала перевести процесс в состояние приостановки (*T*), а затем ввести команду *bg <n>*.

Заметим, что один и тот же скриптовый файл может быть запущен сколько угодно раз. При этом операционная система создаст соответствующее количество процессов. Чтобы управление оставалось за терминалом, скриптовый файл каждый раз следует запускать в фоновом режиме.

Для завершения активного процесса следует нажать сочетание клавиш *CTRL+C*. Для завершения процесса, работающего в фоновом режиме, необходимо сначала перевести его в активный режим, а затем воспользоваться сочетанием клавиш *CTRL+C*.

3. Дескрипторы процессов. Дескриптор каждого процесса находится в файловой системе */proc*. Различные утилиты, «добывающие» информацию о процессах, обращаются именно к этой файловой системе, поскольку она обеспечивает интерфейс к структурам данных ядра. На рис. 5 показан фрагмент содержимого каталога */proc*.

```
main@main-VirtualBox:/proc$ ls -l
total 0
dr-xr-xr-x  9 root      root           0 map 10 09:00  1
dr-xr-xr-x  9 root      root           0 map 12 09:35 10
dr-xr-xr-x  9 nobody    dip           0 map 12 09:35 1012
dr-xr-xr-x  9 whoopsie  whoopsie     0 map 12 09:35 1084
dr-xr-xr-x  9 root      root           0 map 12 09:35 11
dr-xr-xr-x  9 root      root           0 map 12 09:35 12
dr-xr-xr-x  9 root      root           0 map 12 09:35 125
```

Рис. 5. Фрагмент содержимого каталога */proc*

Подкаталоги, имена которых состоят из цифр — это *PID* процессов. Когда операционная система создает процесс, в каталоге */proc* появляется подкаталог с именем, составляющим *PID* этого процесса. При завершении процесса соответствующий ему подкаталог удаляется. Для незамедлительного изменения содержимого файловой системы */proc* физически находится в оперативной памяти.

Также каталог */proc* содержит файлы, в которых находится общая информация о всех процессах, созданных операционной системой. Рассмотрим некоторые из них.

Файл *uptime* содержит общее время работы операционной системы в секундах и идеализированное время, затрачиваемое на один процесс. Оба числа представлены десятичными дробями.

Файл *stat* отображает общую статистику работы операционной системы. Содержимое файла зависит от архитектуры системы (рис. 6.).

```

GNU nano 2.5.3 File: stat
cpu 698034 19924 95991 5774911 61822 0 1824 0 0 0
cpu0 264726 10135 48178 2963313 33160 0 104 0 0 0
cpu1 433307 9789 47813 2811597 28662 0 1720 0 0 0
intr 14835412 33 13418 0 0 0 0 0 0 119 0 0 23152 0 0 32973 0 0 36 79877 14365$
ctxt 25530995
btime 1584046671
processes 9787
procs_running 2
procs_blocked 0
softirq 4090195 15 2390584 6783 87667 142365 0 8935 881120 0 572726

```

Рис. 6. Фрагмент содержимого файла */proc/stat*

Поясним значения некоторых строк.

cpu — количество тиков за время работы системы в (в скобках указаны номера столбцов):

(1) - пользовательском режиме; (2) — пользовательском режиме с низким приоритетом; (3) — в системном режиме; (4) — в режиме идеальной задачи; (5) — ожидании завершения операций ввода-вывода.

intr — количество прерываний, которые произошли с момента загрузки операционной системы (первый столбец).

ctxt — количество переключений контекста.

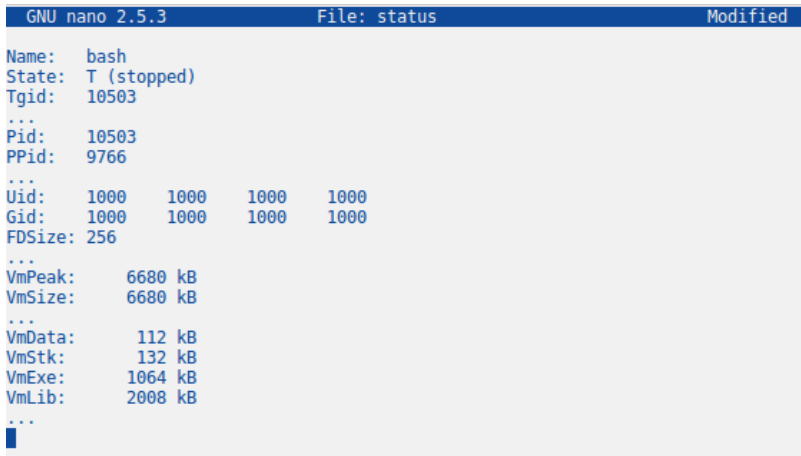
processes — количество процессов с момента загрузки системы.

procs_running — количество процессов, находящихся в состоянии выполнения.

procs_blocked — количество процессов, заблокированных в ожидании завершения операции ввода-вывода.

Запустим скриптовый файл *process* в фоновом режиме и по *PID* найдем **соответствующий ему каталог** в файловой системе */proc*. Рассмотрим содержимое некоторых файлов, содержащих информацию о данном процессе.

Общую информацию о процессе содержит файл *status*. Фрагмент его содержимого приведен на рис. 7.



```

GNU nano 2.5.3      File: status      Modified
Name:  bash
State:  T (stopped)
Tgid:  10503
...
Pid:   10503
PPid:  9766
...
Uid:   1000    1000    1000    1000
Gid:   1000    1000    1000    1000
FDSIZE: 256
...
VmPeak: 6680 kB
VmSize: 6680 kB
...
VmData: 112 kB
VmStk:  132 kB
VmExe:  1064 kB
VmLib:  2008 kB
...

```

Рис. 7. Фрагмент содержимого файла `/proc/<PID>/status`

Поясним значения некоторых строк.

Name — имя утилиты, выполняющей процесс.

State — состояние процесса.

Tgid — идентификатор группы потоков в составе процесса.

Uid — (1) *RUID*, (2) *EUID*.

Gid — (1) *RGID*, (2) *EGID*.

FDSIZE — количество файловых дескрипторов.

VmData, *VmStk*, *VmExe* — размеры сегмента данных, стека и текстового сегмента (кода) процесса.

Символическая ссылка `/proc/<PID>/cwd`, указывает на рабочий каталог процесса, т. е. каталог, в котором находится сам исполняемый файл.

4. Сессии, группы процессов и управляющий терминал. При работе в операционной системе посредством терминала для удобного управления процессами используются абстракции «группа процессов», «сессии» и «управляющий терминал».

Группа процессов — коллекция одного или нескольких процессов. Цель объединения процессов в группы — упрощение отправки сигналов таким процессам: один сигнал может прервать, приостановить или продолжить работу всех процессов в группе. Каждая группа процессов обладает *идентификатором группы процессов (PGID)* и имеет *лидера группы* — процесс, чей *PID* равен идентификатору группы процессов. Команда `jobs` выводит на экран

порядковый номер процесса-лидера группы. Создадим в домашнем каталоге (~) скриптовый файл *process_group*, при запуске которого будут созданы два процесса, принадлежащие одной группе (рис. 8).

```

GNU nano 4.8                                process_group
#!/bin/bash
cd ~
echo $$
./process
  
```

Рис. 8. Скриптовый файл, создающий группу процессов

Набор из одной или нескольких групп процессов образует *сессию*. Каждая сессия обладает *идентификатором сессии (SID)*. Сессии необходимы для установления порядка действий среди авторизованных в системе пользователей. Когда пользователь впервые входит в систему, процесс авторизации *login* предлагает ввести имя учетной записи и соответствующий пароль. В случае успешной авторизации создается сессия, лидером которой является процесс *login*. *Лидер сессии* — процесс, *PID* которого равен идентификатору сессии.

ЗАДАНИЯ К ЛАБОРАТОРНОЙ РАБОТЕ

1. Изучить теоретический материал.
2. Авторизоваться в терминале *tty1*.
3. Вывести снимок состояний процессов, запущенных пользователем.
4. Вывести снимок состояний процессов, запущенных с текущего терминала.
5. Вывести список состояний процессов в режиме реального времени с полями, указанными в таблицах ниже:

5.1

<i>USER</i>	<i>PID</i>	<i>PPID</i>	<i>UID</i>	<i>RUID</i>	<i>SUID</i>	<i>RUSER</i>	<i>SUSER</i>	<i>TTY</i>
-------------	------------	-------------	------------	-------------	-------------	--------------	--------------	------------

5.2

<i>%CPU</i>	<i>S</i>	<i>DATA</i>	<i>P</i>	<i>CODE</i>	<i>nMAJ</i>	<i>ENVIRON</i>	<i>nMIN</i>	<i>%MEM</i>
-------------	----------	-------------	----------	-------------	-------------	----------------	-------------	-------------

6. На справочных страницах утилиты *top* найти информацию о значении всех полей таблиц предыдущего задания.
7. Определить *PPID* процессов, идентификаторы которых равны *PID=1*, *PID=15*, *PID=100* .
8. Создать в **домашнем каталоге** с помощью текстового редактора *nano* скриптовый файл с именем *process*, как показано в п. 2. Выполнить следующую последовательность действий:

- 8.1 Запустить скриптовый файл в активном режиме;
 8.2 Перевести выполнение скриптового файла в фоновый режим;
 8.3 Вывести на экран в режиме реального времени (утилита *top*) указанную в таблице ниже информацию о процессе, соответствующему запущенному скриптовому файлу:

<i>USER</i>	<i>PID</i>	<i>PR</i>	<i>S</i>	<i>NI</i>	<i>%CPU</i>	<i>%MEM</i>
-------------	------------	-----------	----------	-----------	-------------	-------------

- 8.4 Перевести выполнение скриптового файла в активный режим;
 8.5 Перевести выполнение скриптового файла в состояние приостановки (*T*);
 8.6 Перевести выполнение скриптового файла в состояние выполнения (*R*) в активном режиме;
 8.7 Завершить выполнение скриптового файла. Убедиться, что соответствующий процесс завершен.
9. Продемонстрировать умение запускать скриптовый файл на выполнение с помощью команд *bash* и «.».
10. Запустить скриптовый файл *process* в фоновом режиме три раза. Продемонстрировать умение работать с утилитами *jobs*, *fg*, *bg*.
11. Завершить выполнение всех трех процессов. Убедиться, что все процессы, соответствующие скриптовому файлу *process*, завершены.
12. Используя файловую систему */proc*:
- 12.1 Определить общее время работы операционной системы в секундах и минутах.
- 12.2 Определить количество процессов в системе, находящихся в состоянии выполнения (*R*).
- 12.3 Определить количество процессов в системе, находящихся в состоянии прерываемого ожидания (*S*).
- 12.4 Определить общее количество процессов в операционной системе.
- 12.5 Определить рабочий каталог процесса, идентификатор которого равен 1(*PID=1*).
- 12.6 Для любых трех процессов пользователя заполнить таблицу ниже.

<i>PID</i>	<i>PPID</i>	<i>STATE</i>	<i>RUID</i>	<i>EUID</i>	<i>VM DATA</i>	<i>VMSTK</i>	<i>VMEXE</i>

- 12.7 Определить *PID* **всех** процессов в операционной системе,

находящихся в состоянии выполнения (*R*) и заполнить таблицу ниже.

<i>PID</i>	<i>PPID</i>	<i>STATE</i>
		R
...	...	R

- 12.8 Выполнить команды *bash process&* и *sudo bash process&*. Пояснить различие строк *Uid* и *Gid* файлов */proc/<PID>/status*, соответствующих запущенным процессам.
13. Определить *PID* процесса-лидера сессии для пользователя, авторизованного в терминале *tty1*.
14. Создать в **домашнем каталоге** скриптовый файл *process_group*, как показано на рис. 8 и запустить его на выполнение в фоновом режиме:
- 14.1 Определить *PID* всех процессов, входящих в группу, лидером которой он является (утилита *ps jf*).
- 14.2 Перевести в активный режим процесс-лидера сессии и завершить его выполнение сигналом *SIGINT*.
15. Авторизоваться в терминалах *tty2* и *tty3*. Определить *PID* процессов-лидеров соответствующих сессий

ЛАБОРАТОРНАЯ РАБОТА № 2

ИЗУЧЕНИЕ МЕХАНИЗМОВ ВЗАИМОДЕЙСТВИЯ ПРОЦЕССОВ В LINUX

1. IPC. Каждый процесс в операционной системе выполняется в собственном адресном пространстве. Для обмена данными между процессами в пределах одной операционной системы, или между процессами, выполняющимися на разных системах, стандартом POSIX предусмотрен механизм межпроцессного взаимодействия *IPC* — *Inter-Process Communication*. *IPC* представляет собой набор методов для обмена данными между потоками процессов.

Две фундаментальные модели межпроцессного взаимодействия — отправка сообщений (а) и разделяемая память (b) — показаны на рис. 1.

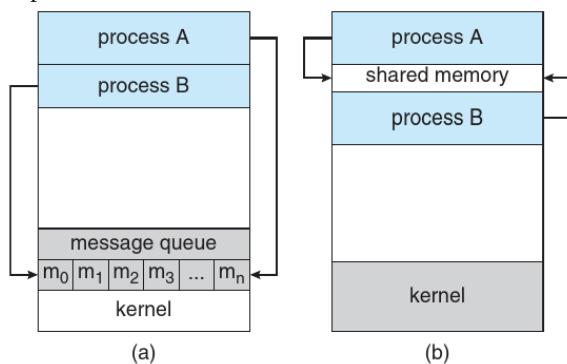


Рис. 1. Модели *IPC*

Далее рассмотрим некоторые средства, реализующие механизм *IPC*.

2. Сигналы. Сигнал представляет собой простейшее средство межпроцессного взаимодействия, реализующее внешнее управление процессами. *Сигнал* — это программное прерывание. Сигналы способны в любое время прерывать процесс для обработки какого-либо события. Процесс может быть прерван сигналом по инициативе другого процесса или ядра операционной системы. Последнее использует сигналы для извещения процессов о различных событиях, например о завершении дочернего процесса.

Использование сигнала состоит из следующих этапов:

- генерирование сигнала;

- доставка сигнала процессу;
- обработка полученного сигнала.

Обработка полученного сигнала осуществляется либо обработчиком по умолчанию, либо обработчиком, определенным пользователем. Каждый сигнал имеет обработчик по умолчанию, который запускается ядром операционной системы при его обработке. При получении сигнала процесс прерывается, управление передается обработчику сигнала. Если функция обработки сигнала не приводит к завершению процесса, то управление передается в точку, на которой процесс был прерван. Также процесс может игнорировать сигнал.

Генерирование сигналов может осуществляться посредством отправки с терминала так называемых *управляющих символов*, которые образуются при нажатии сочетаний клавиш (*CTRL+C*, *CTRL+* и др.), а также с использованием утилиты *kill*. Обычно процессам, находящимся в активном режиме, сигналы генерируются сочетанием клавиш, а фоновым процессам — с помощью утилиты *kill*.

ОС *Linux* поддерживает работу с сигналами. Общее их количество около 60. Некоторые сигналы и их назначение приведены в таблице ниже.

Сигнал	Код	Описание
<i>Сигналы завершения</i>		
<i>SIGHUP</i>	1	Сигнал, информирующий о потере соединения с управляющим терминалом
<i>SIGINT</i>	2	Сигнал штатного завершения процесса (генерируется сочетанием клавиш <i>CTRL+C</i>)
<i>SIGQUIT</i>	3	Сигнал штатного завершения процесса (генерируется сочетанием клавиш <i>CTRL+\</i>). Создается дамп памяти процесса
<i>SIGKILL</i>	9	Сигнал безусловного завершения процесса (возможна потеря данных)
<i>SIGTERM</i>	15	Сигнал штатного завершения процесса. Отправляется процессам, не имеющим управляющего терминала
<i>Сигналы управления</i>		
<i>SIGCHLD</i>	17	Сигнал изменения статуса дочернего процесса. По умолчанию игнорируется родительским

		процессом
<i>SIGSTOP</i>	19	Сигнал приостановки выполнения процесса
<i>SIGSTP</i>	20	Сигнал приостановки выполнения процесса (генерируется сочетанием клавиш <i>CTRL + Z</i>)
<i>SIGCONT</i>	18	Сигнал продолжения выполнения приостановленного процесса
<i>Другие сигналы</i>		
<i>SIGALRM</i>	14	Сигнал, посылаемый процессу по истечении заданного времени
<i>SIGPWR</i>	30	Сигнал, посылаемый при аварии сетевого питания

Сигналы завершения используются для сообщения о завершении процесса и для того, чтобы работающая программа выполнила некоторые действия перед завершением. Заданное по умолчанию действие для сигналов завершения — завершение выполнения процесса. В некоторых случаях обработчик не только завершает процесс, но и создает дампы памяти для последующей отладки.

Некоторые варианты использования утилиты *kill* приведены в таблице.

Команда [-опции]	Действие	Реакция процесса
<i>kill -l</i>	Вывод на экран списка возможных сигналов и их номеров	—
<i>kill -2 <PID></i>	Посылка сигнала <i>SIGINT</i> процессу с идентификатором <i>PID</i>	Завершение
<i>kill -9 <PID_1> <PID_2></i>	Посылка сигнала <i>SIGKILL</i> двум процессам с <i>PID_1</i> и <i>PID_2</i>	Завершение
<i>kill -18 <PID></i>	Посылка сигнала <i>SIGCONT</i> процессу с идентификатором <i>PID</i>	Возобновление работы
<i>kill -19 <PID></i>	Посылка сигнала <i>SIGSTOP</i>	Остановка

	процессу с идентификатором <i>PID</i>	
--	--	--

В качестве примера с одного терминала дважды запустим исполняемый файл *process* в фоновом режиме, а затем отправим соответствующим процессам сигналы *SIGTERM* и *SIGSTOP* (рис. 2).

```

PID TTY      STAT   TIME COMMAND
1493 pts/1    Ss     0:00  bash
2490 pts/1    R      1:12  bash process
2491 pts/1    R      0:33  bash process
2492 pts/1    R+     0:00  ps -t
main@main-Ubuntu:~$ kill -15 2490
main@main-Ubuntu:~$ kill -19 2491
[1]  Завершено      bash process

[2]+  Остановлен    bash process
main@main-Ubuntu:~$ ps -t
PID TTY      STAT   TIME COMMAND
1493 pts/1    Ss     0:00  bash
2491 pts/1    T      1:15  bash process
2493 pts/1    R+     0:00  ps -t

```

Рис. 2. Отправка сигналов процессам

Процесс с *PID*=2490, получив сигнал *SIGTERM*, завершил работу, а процесс с *PID*=2491, получив сигнал *SIGSTOP* приостановил свою работу. Для возобновления работы приостановленного процесса необходимо отправить сигнал *SIGCONT* (рис. 3).

```

main@main-Ubuntu:~$ kill -18 2491
main@main-Ubuntu:~$ ps -t
PID TTY      STAT   TIME COMMAND
1493 pts/1    Ss     0:00  bash
2491 pts/1    R      1:19  bash process
2496 pts/1    R+     0:00  ps -t

```

Рис. 3. Возобновление работы приостановленного процесса

Процесс может игнорировать (*IGNORED*), перехватить (*CAUGHT*), блокировать (*BLOCKED*) или ожидать доставку сигнала. Диспозицию сигналов можно узнать командой *ps -s*. На рис. 4 показана диспозиция сигналов для процессов, запущенных с терминала.

```

main@main-Ubuntu:~$ ps -s
  UID   PID  PENDING  BLOCKED  IGNORED  CAUGHT  STAT  TTY      TIME  COMMAND
 1000  1493  00000000  00010000  00380004  4b817efb  Ss    pts/1    0:00  bash
 1000  2491  00000000  00000000  00000000  00010000  R      pts/1    3:01  bash proc
 1000  2515  00000000  00000000  00000000  <f3d1fef9  R+     pts/1    0:00  ps -s

```

Рис. 4. Диспозиция сигналов

Битовые маски представлены в шестнадцатеричной системе счисления, где каждый *n*-й бит (счет разрядов идет справа налево)


```

main@main-Ubuntu:~$ ps -x|grep /usr/bin
 1042 ?      Ssl    0:00 /usr/bin/pulseaudio --daemonize=no --log-target=journal
 1045 ?      Sl     0:00 /usr/bin/gnome-keyring-daemon --daemonize --login
 1051 ?      Ss     0:06 /usr/bin/dbus-daemon --session --address=systemd: --nof
ork --nopidfile --systemd-activation --syslog-only
 1132 ?      Ss     0:00 /usr/bin/ssh-agent /usr/bin/im-launch startxfce4
 1162 ?      S      0:00 /usr/bin/dbus-daemon --config-file=/usr/share/defaults/
at-spi2/accessibility.conf --nofork --print-address 3
 1182 ?      Sl     0:00 /usr/bin/xfce4-screensaver --no-daemon
 1443 ?      S      0:00 /usr/bin/python3 /usr/share/system-config-printer/apple
t.py
 1448 ?      Sl     0:00 /usr/bin/python3 /usr/bin/blueman-applet
 1474 ?      Ssl    0:01 /usr/bin/spice-vdagent
 1539 ?      Ssl    0:04 /usr/bin/Thunar --daemon
 1555 ?      Sl     0:00 /usr/bin/python3 /usr/bin/blueman-tray
 4001 pts/1  S+     0:00 grep --color=auto /usr/bin

```

Рис. 6. Использование неименованного канала

Именованные каналы позволяют обмениваться информацией неродственным процессам. Два любых процесса могут организовать однонаправленную передачу информации посредством файла именованного канала с заранее согласованным именем. Файл канала должен быть создан до начала взаимодействия процессов.

Для создания файла именованного канала служит утилита *mkfifo* <имя_файла_канала>. Файл именованного канала отличается от обычного (не специального) файла тем, что с ним могут одновременно работать два процесса: один процесс может выполнять операцию записи, а другой — операцию чтения.

Рассмотрим пример, демонстрирующий работу канала. Создадим именованный канал с именем *pipe*. В терминале *tty1* выполним команду *cat pipe*, считывающей содержимое канала. Пока канал пуст, утилита *cat* ничего не выведет на экран. Авторизуемся в терминале *tty2* и выполним команду *ps -x > pipe*, которая направит результат работы команды *ps -x* в файл *pipe*. Как только завершится работа этой команды на экране первого терминала появится ее результат (рис. 7). Переключение между терминалами происходит при нажатии сочетания клавиш *CTRL+ALT+F1* — *CTRL+ALT+F2*.

```

main@main-Ubuntu:~$ mkfifo pipe
main@main-Ubuntu:~$ ls -l pipe
prw-rw-r-- 1 main main 0 окт  1 14:16 pipe

```

а)

```

main@main-Ubuntu:~$ tty
/dev/pts/0
main@main-Ubuntu:~$ ps -x > pipe

```

б)

```

main@main-Ubuntu:~$ tty
/dev/pts/1
main@main-Ubuntu:~$ cat pipe
  PID TTY          STAT       TIME COMMAND
 1121 ?        Ss          0:00    /lib/systemd/systemd --user
 1122 ?        S           0:00    (sd-pam)
 1140 ?        S<sl        0:00    /usr/bin/pulseaudio --daemonize=no --log-target=jou
rnal

```

в)

Рис. 7. Использование именованного канала: создание канала (а), передача данных в канал (б), чтение из канала (в)

В данном примере стандартный поток вывода команды `ps -x` был перенаправлен в именованный канал `pipe`, после чего утилита `cat` прочитала содержимое канала и вывела его на экран.

ЗАДАНИЯ К ЛАБОРАТОРНОЙ РАБОТЕ

1. Изучить теоретический материал.
2. Авторизоваться в терминалах `tty1` и `tty2`.
3. Переключиться на терминал `tty1`.
4. Отправка сигналов процессам:
 - 4.1. Запустить скриптовый файл `process` три раза в фоновом режиме.
 - 4.2. Вывести на экран `PID` созданных процессов.
 - 4.3. Первому процессу отправить сигнал `SIGSTOP`, а второму — `SIGTERM`. Убедиться, что сигналы получены соответствующими процессами.
 - 4.4. Третьему процессу отправить сигнал `SIGKILL`.
 - 4.5. Отправить сигнал `SIGCONT` второму процессу.
 - 4.6. Отправить сигнал `SIGKILL` второму процессу.
 - 4.7. Проверить, что работа всех трех процессов завершена.
5. Перехват сигналов:
 - 5.1. В домашнем каталоге создать скриптовый файл `proc_trap`, как показано на рис. 5.
 - 5.2. Запустить созданный скрипт в активном режиме.
 - 5.3. Отправить ему сигнал `SIGINT`.
 - 5.4. Пояснить реакцию выполняющегося процесса на отправленный сигнал.
 - 5.5. Модифицировать скрипт `proc_trap` так, чтобы при получении им сигнала `SIGINT` на терминал выводился путь текущего каталога. Для выполнения этого задания необходимо обратиться к справочным страницам утилиты `trap`.

- 5.6. Модифицировать скрипт *proc_trap* так, чтобы при получении им сигнала *SIGTERM* запускался бы на выполнение скриптовый файл *process*. Для выполнения этого задания необходимо обратиться к справочным страницам утилиты *trap*.
- 5.7. Результаты выполнения заданий 5.5 и 5.6 пояснить.
6. Диспозиция сигналов:
- 6.1. Вывести на экран диспозицию сигналов всех процессов, запущенным пользователем.
- 6.2. Для **двух** любых процессов операционной системы определить в **десятичной системе счисления** коды игнорируемых, перехватываемых, блокируемых и ожидающих доставку сигналов и заполнить таблицу ниже:

<i>PID</i>	<i>Игнорируемые сигналы</i>	<i>Перехватыва емые сигналы</i>	<i>Заблокирован ные сигналы</i>	<i>Ожидающие доставку сигналы</i>

7. Каналы:
- 7.1. Повторить демонстрацию работы неименованного канала из п. 3. Пояснить работу конвейера.
- 7.2. Создать в домашнем каталоге файл именowanego канала *pipe*.
- 7.3. Повторить демонстрацию работы именowanego канала из п. 3.
- 7.4. В терминале *tty1* выполнить команду *cat pipe*. В терминале *tty2* выполнить команду *ls -l -R / >pipe*. Переключиться несколько раз между терминалами *tty1* и *tty2*. Пояснить работу именowanego канала *pipe*.

ЛАБОРАТОРНАЯ РАБОТА № 3

ИЗУЧЕНИЕ РАБОТЫ ПЛАНИРОВЩИКА LINUX

1. Общие сведения о планировании. Одной из важных функций, выполняемых ядром операционной системы, является функция планирования процессов. *Системный планировщик* — компонент ядра операционной системы, занимающийся переключением центрального процессора между процессами.

Процессы условно можно разделить на интенсивно использующие операции ввода-вывода (*I/O-bound*), и интенсивно использующие центральный процессор (*processor-bound*). Процессы первого типа не сильно загружают центральный процессор и находятся в основном в состоянии ожидания завершения операций ввода-вывода. Процессы второго типа сильно нагружают центральный процессор, большую часть времени выполняя программный код. Существуют и такие процессы, которые на разных стадиях выполнения могут относиться как к первому, так и ко второму типам.

Операционная система *Linux* классифицирует процессы на «обычные» и процессы *реального времени*. Процессы реального времени всегда пользуются преимуществом по отношению к обычным процессам.

Очередь готовых на выполнение процессов может быть большой и для эффективного использования процессорного времени планировщик использует различные *алгоритмы планирования*, оперирует *приоритетами* и *классами приоритетов* процессов, а также использует *кванты времени*.

В традиционных *UNIX* системах алгоритмы планирования основаны на квантах времени. *Квант* — это число, отражающее длительность выполнения процесса до его вытеснения другим процессом. По завершению кванта процессор будет переключен на другой процесс.

Алгоритм планирования *CFS (Completely Fair Sheduler)*, использует так называемую долю процессорного времени. Такой алгоритм планирования хорошо подходит для «обычных» процессов *Linux*. Для организации планирования используется значение *nice*.

Алгоритм планирования *FIFO (First In First Out)*, используется для планирования процессов реального времени. Данный алгоритм позволяет процессу выполняться непрерывно долго, до момента перехода в состояние сна. При появлении в системе готового к

выполнению процесса с более высоким приоритетом, планировщик переключает процессор на этот процесс.

Алгоритм планирования *RR (Robin Round)*, аналогичен алгоритму *FIFO* с той лишь разницей, что содержит дополнительные правила для обслуживания процессов равного приоритета.

2. Планирование «обычных» процессов. В ОС *Linux* для организации планирования «обычных» процессов используется так называемый *относительный приоритет* — включает диапазон значений параметра *nice* (от англ. *nice* — *тактичный*) от -20 до +19. Процессам с низким значением параметра *nice* выделяется больше процессорного времени.

Продемонстрируем это на простом примере.

Чтобы искусственно создать конкуренцию двух процессов за центральный процессор, необходимо выполнить привязку каждого процесса к одному и тому же процессору (ядру центрального процессора). Для этого сначала определим количество ядер центрального процессора командой *lscpu* (выделенная строка на рис. 1).

GNU nano 4.8		cpu
Архитектура:		x86_64
CPU op-mode(s):		32-bit, 64-bit
Порядок байт:		Little Endian
Address sizes:		42 bits physical, 48 bits virtual
CPU(s):		2
On-line CPU(s) list:		0,1
Потоков на ядро:		1

Рис. 1. Информация о количестве ядер процессора

Далее в фоновом режиме дважды запустим скриптовый файл *process* и выполним привязку обоих процессов к ядру 1 (рис. 2).

```
main@main-Ubuntu:~$ taskset -p -c 1 1987
pid 1987's current affinity list: 0,1
pid 1987's new affinity list: 1
main@main-Ubuntu:~$ taskset -p -c 1 1988
pid 1988's current affinity list: 0,1
pid 1988's new affinity list: 1
```

Рис. 2. Привязка процессов к одному ядру центрального процессора

Команда *taskset* производит привязку указанных по *PID* процессов к конкретному центральному процессору (ядру процессора) как показано в таблице.

Команда [-опции]	Действие
<i>taskset -p -c <n> <PID></i>	Привязка процесса с <i>PID</i> к процессору (ядру процессора) с номером <i>n</i> (<i>n=0,1,2...</i>)

Значения *nice*, установленные для запущенных процессов операционной системой по умолчанию, равны нулю (рис. 3). При этом и процессорное время между ними распределено равномерно.

PID	USER	PR	NI	S	%CPU	TIME+	COMMAND
1987	main	20	0	R	50,2	9:03.42	process
1988	main	20	0	R	49,8	9:01.50	process

Рис. 3. Значения *nice*

Чтобы изменить это значение служит утилита *renice*. Некоторые варианты использования утилиты *renice* приведены в таблице.

Команда [-опции]	Действие
<i>renice -n 5 -p <PID></i>	Установка значения <i>nice</i> =5 для процесса с идентификатором <i>PID</i>
<i>renice -n -2 -p<PID></i>	Установка значения <i>nice</i> =-2 для процесса с идентификатором <i>PID</i>

Увеличим значение *nice* для одного из процессов на значение 10 и посмотрим распределение процессорного времени (рис. 4. а).

Уменьшим значение *nice* для того же процесса до значения -5 и посмотрим распределение процессорного времени (рис. 4. б).

PID	USER	PR	NI	S	%CPU	TIME+	COMMAND
1988	main	20	0	R	90,0	13:47.28	process
1987	main	30	10	R	9,6	12:58.64	process

а)

PID	USER	PR	NI	S	%CPU	TIME+	COMMAND
1987	main	15	-5	R	74,8	14:31.27	process
1988	main	20	0	R	24,6	23:44.33	process

б)

Рис. 4.

Понижать значение *nice* (увеличивать приоритет процесса) может только суперпользователь.

3. Планирование процессов реального времени. Второй вид — *абсолютный приоритет* (приоритет реального времени) — включает диапазон чисел от 0 до 99. Большему значению приоритета реального времени соответствует больший приоритет.

Для процессов реального времени операционная система *Linux* использует два алгоритма планирования: *FIFO* (*First In First*

Out) и *RR* (*Round Robin*). Алгоритм *RR* реализует циклическое планирование с фиксированными квантами времени. Оба алгоритма при работе организуют процессы в одну приоритетную очередь. На выполнение выбирается готовый процесс с большим приоритетом.

ЗАДАНИЯ К ЛАБОРАТОРНОЙ РАБОТЕ

1. Изучить теоретический материал.
2. Авторизоваться в терминалах *tty1* и *tty2*.
3. Переключиться на терминал *tty1*.
4. Найти в операционной системе процессы с различными абсолютными и относительными приоритетами и заполнить таблицу (утилиты *top*, *ps -ax*)

№	PID	PRI (абсолютный приоритет)	NI (относительный приоритет)
1			
2			
...			
5			

5. Относительный приоритет процесса:
 - 5.1. Запустить скриптовый файл *process* два раза в фоновом режиме.
 - 5.2. Выполнить привязку соответствующих процессов к ядру №1 центрального процессора.
 - 5.3. В терминале *tty2* выполнить команду *top -u <имя_пользователя>*.
 - 5.4. Для любого из двух запущенных процессов изменять значения *nice* от -10 до +10 с шагом 2 и заполнить таблицу ниже:

NICE	-10	-8	...	10
PID_1				
%CPU			...	
PID_2				
%CPU			...	

- 5.5. Завершить выполнение скриптовых файлов.
6. Абсолютный приоритет процесса:
 - 6.1. Используя страницы руководства *man*, изучить работу утилит *chrt*.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Oracle VM VirtualBox, User manual. – Version 5.1.26. – 2017.
2. Забродин Л.Д. UNIX: основы командного интерфейса и программирования (в примерах и задачах) / Л.Д. Забродин, В.В. Макаров, А.Б. Вавренюк, – М.: Национальный исследовательский ядерный университет «МИФИ», 2010.
3. Костромин В.А. Linux для пользователя / В.А. Костромин. – СПб.: БХВ-Петербург, 2003.
4. Abraham Silberschats Operating System Concepts, 2013.
5. Кетов Д.В. Внутреннее устройство Linux. – Спб.: БВХ – Петербург, 2017. – 320 с.: ил.

Изучение процессов в LINUX

Составители: Бубнов Сергей Алексеевич
Бубнов Алексей Алексеевич

Редактор Р.К. Мангутова
Корректор С.В. Макушина

Подписано в печать 27.06.21. Формат бумаги 60х84 1/16.

Бумага писчая. Печать трафаретная. Усл. печ. л. 1,75

Тираж 50 экз. Заказ

Рязанский государственный радиотехнический университет.

390005, Рязань, ул. Гагарина, 59/1.

Редакционно-издательский центр РГРТУ.