

Московский государственный технический университет
имени Н.Э. Баумана

(национальный исследовательский университет)

Факультет «Информатика и системы управления»

Кафедра «Компьютерные системы и сети»

В.Ю. Мельников, Е.К. Пугачев

Исследование способов организации оперативной памяти и взаимодействия процессов

Электронное учебное издание

Методические указания по выполнению лабораторных работ

по дисциплине "Операционные системы"

2017

Цель работы: получение теоретических и практических сведений об управлении процессами, потоками и оперативной памятью в UNIX-подобных системах и в Linux в частности.

Время выполнения - 4 часа.

Организация оперативной памяти.

Оперативная память - это энергозависимая часть компьютерной памяти в которой во время работы компьютера хранятся программы и их рабочие данные.

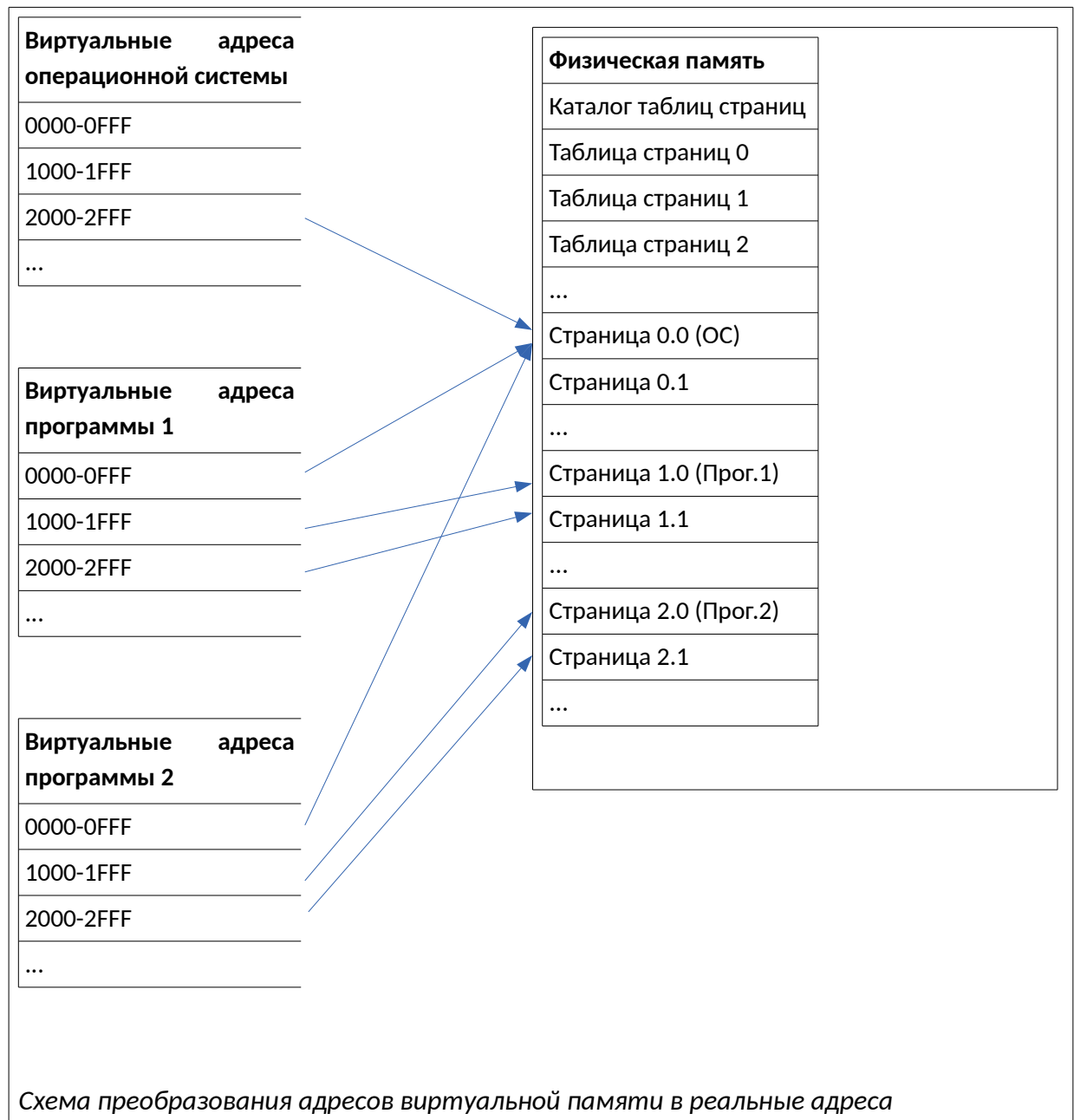
Для выполнения программы, её необходимо загрузить в оперативную память. Программы не могут работать с данными, находящимися непосредственно на жёстком диске, их сначала надо загрузить в оперативную память. А это длительная операция. Поэтому, обычно объём оперативной памяти является критически ресурсом увеличение которого резко ускоряет работу программ, причём гораздо сильнее, чем увеличение быстродействия процессора. Но, оперативная память дорогая, как по деньгам, так и по энергозатратам. На её поддержание постоянно требуется расходовать энергию, что сказывается на времени автономной работы планшетов и ноутбуков. Так что, оперативной много не бывает.

Защита памяти

Ранние версии ОС использовали реальный режим адресации оперативной памяти при котором адреса, используемые программами соответствовали физическим адресам оперативной памяти. В этом режиме, программы имели неограниченный доступ к областям памяти друг друга и могли намеренно или случайно повредить их. Исключительно благоприятная среда для распространения вирусов.

В современных ОС каждый процесс выполняется в виртуальной памяти - виртуальном адресном пространстве, которое аппаратно проецируется на адреса реальной памяти, выделенной программе. При попытке обращения к адресам за её пределами, вызывается прерывание "ошибка страницы" и операционная система прерывает выполнение программы. Для обмена данными между процессами используются области коллективного доступа. Ядро ОС помещается в адресное пространство программы, но помечается как недоступное для пользователя.

Рассмотрим как работает виртуальная память:



Вся физическая память делится на страницы фиксированного размера (обычно 4 КБайт).

Виртуальные адреса (которыми оперирует программа) делятся на 2 части:

- номер виртуальной страницы
- смещение в рамках страницы — биты 11-0 (12 бит).

Для каждого процесса операционная система создаёт таблицу соответствия виртуальных страниц реальным страницам. (Формат таблиц зависит от типа процессора, часто используются многоуровневые таблицы)

При обращении программы к памяти виртуальный адрес аппаратно пересчитывается в реальный согласно этой таблице.

Для программы виртуальная память выглядит, как непрерывное адресное пространство, в пределах выделенного лимита, вне зависимости от наличия у компьютера такого количества оперативной памяти. При обращении к виртуальной странице, для которой не хватает реальной памяти, вызывается прерывание "ошибка страницы"(page fault). Операционная система сначала выгружает на диск страницу, к которой давно не обращались, а затем заносит в таблицу страниц ссылку на освободившуюся страницу и возвращает управление прерванной программе. Данный процесс называется подкачкой или свопингом.

В ОС Windows содержимое памяти сохраняется в файл XXXXXX.sys, в linux используется раздел подкачки.

Но надо понимать, что на выгрузку и подгрузку страниц с диска уходит много времени и при серьёзной нехватке памяти свопинг не спасает. Система начинает работать так медленно, что даже завершение работы лишних программ становится проблемой. Система начинает работать так медленно, что реакция на простейшие действия наступает через несколько секунд. В таком случае, особенно в графическом интерфейсе, надо на некоторое время прекратить нажимать кнопки, дожидаться выполнения тех действий, которые вы уже сделали и только после этого продумывать каждое действие и дожидаться на него реакции. При возникновении свопинга на сервере, подключиться к нему в текстовом интерфейсе и прервать выполнение некоторых программ. Затем изменить настройки программ так, чтобы они использовали меньше памяти. Например, уменьшить размер кеша таблиц и индексов для СУБД.

Отображаемые в память файлы

Хотелось бы упомянуть ещё один приём, использования виртуальной памяти - отображаемые в память файлы. Представьте себе, что запускается система управления базой данных (СУБД). При этом надо прочесть большой объём данных (индексы, и описания таблиц), часто превышающий объём физической памяти. При этом часть данных выгружается в файл подкачки, что ещё увеличивает время запуска. Программа может самостоятельно подгружать эти данные по мере необходимости, а может отобразить файл базы данных в память и его будет подгружать, по необходимости, менеджер виртуальной памяти. Запуск СУБД в этом случае будет очень быстрым, но первое время СУБД будет работать медленно, пока не загрузятся часто используемые данные.

Кеширование операций чтения и записи на диск

Итак, в памяти у нас хранится ядро операционной системы, выполняемые программы и их данные. Но это ещё не всё. Значительную часть дефицитной оперативной памяти в памяти занимает кэш дисковых операций. Разберёмся, из чего он состоит и для чего он нужен.

Страничный кеш — сюда попадает всё, что вы читали и записывали на диск. Когда программе снова потребуется содержимое файла, который только что записала другая программа, она его моментально получит. Прежде, чем читать файл с диска, ОС пытается найти его в кэше. Если какой-то файл недавно читали, второй раз он будет быстро прочитан из памяти. Этот вид кэша занимает больше всего места в памяти. Вы не можете явно указать сколько мегабайт может система использовать под кэш, но можно настроить скорость удаления просроченных страниц из кэша. В Linux файл `/proc/sys/vm/vfs_cache_pressure` содержит число, которое говорит, насколько агрессивно нужно удалять страницы из кэша. По умолчанию установлен параметр 100. Если его уменьшить ядро будет реже удалять страницы и это приведет к увеличению кэша. При нуле страницы вообще не будут удаляться. Если значение больше 100 страницы будут сразу удаляться быстрее, но чтение замедлится.

Посмотреть размер страничного кэша можно командой `free -h`

	total	used	free	shared	buffers	cached
Mem:	11G	4,7G	6,9G	204M	243M	2,5G
-/+ buffers/cache:		2,0G	9,7G			
Swap:	2,0G	0B	2,0G			

Размер страничного кэша указан в колонке `cached`

Кэш INode — Здесь хранится таблица расположения файлов и папок. Этот кэш невелик, но очень сильно ускоряет работу с файловой системой. Его настройки лучше не менять.

Есть один эффект кэша, который надо учитывать. После того, как программа записала данные в файл, они не сразу записываются на диск. А в параллельном процессе, с небольшой отсрочкой. Это уменьшает нагрузку на диск при записи малыми порциями. Ведь для записи даже одного байта, надо прочитать и целиком перезаписать целый блок не менее 512 байт (обычно несколько Кб).

Программы отсрочки не замечают, поскольку получают данные из кэша и единственное тонкое место этой технологии — аварийное выключение компьютера. Последние данные из кэша не успеют записаться, файлы будут повреждены. Наиболее неприятные последствия потери кэша при записи таблицы расположения файлов и каталогов. Для ранних версий файловых систем аварийное выключение могло привести к повреждению операционной системы вплоть до невозможности загрузки. В современных системах предусмотрено дублирование информации и процедура восстановления структуры файловой системы после аварийного завершения, но она может быть продолжительной. Поэтому надо корректно завершать работу ОС.

Впрочем, в современных ОС нажатие кнопки питания просто запускает процедуру завершения работы, которая посылает сигнал завершения всем запущенным процессам, затем, после их завершения сохраняет данные из кэша и только после этого, даёт команду отключения питания. Если спустя пару минут после нажатия кнопки выключения питания

компьютер не выключился, придётся отключить его принудительно. Для этого надо нажать кнопку питания и не отпускать в течении 3-5 секунд. Компьютер аппаратно выключится.

Кэширование применяется и при записи данных на флешку. Поэтому нельзя сразу выдёргивать флешку, как только закончится копирование файлов. Если у флешки есть световой индикатор надо дождаться пока он прекратит мигать. А ещё лучше использовать команду безопасного отключения носителя в операционной системе или файловом менеджере.

Некоторые модели реализации многопоточности.

В данном разделе даны теоретические сведения о следующих моделях реализации многопоточности в ОС: реализация потоков в пользовательском пространстве, реализация потоков в ядре, гибридная реализация. Курсивом выделены сведения и задания, для дополнительной проработки. Данная работа может выполняться как из графической среды, так и из консольной.

Реализация потоков в пользовательском пространстве.

При данном подходе весь набор потоков размещается в пространстве пользователя, ядро не имеет к ним прямого доступа, оно управляет пользовательскими процессами (содержащими в себе потоки) как обычными однопоточными процессами. При таком подходе многопоточность может быть реализована даже в ОС, не поддерживающих её, с помощью дополнительных библиотек. Ярким примером могут являться процессы и потоки в виртуальной машине: с точки зрения ОС виртуальная машина представляет собой обычный процесс, однако виртуальная машина содержит свой собственный планировщик. Ещё одно преимущество – быстрое переключение контекстов за счёт отсутствия блокировок ядра, каждый процесс обладает собственной таблицей потоков. Пример показан на рисунке 1.

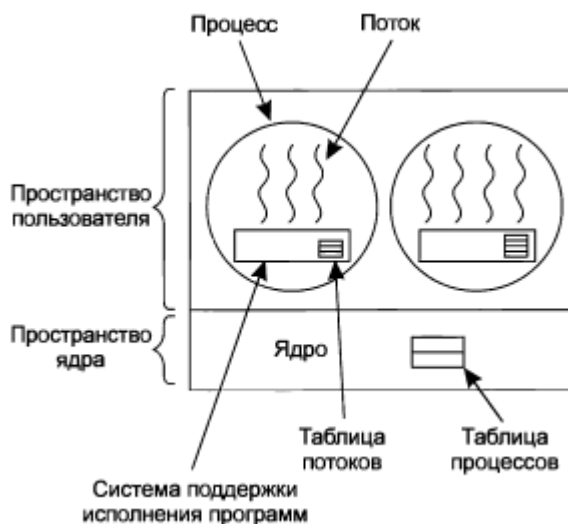


Рисунок 1 – Реализация потоков в пользовательском пространстве.

Несмотря на достоинства, у данного подхода существуют некоторые проблемы:

1. Реализация блокирующих системных вызовов – например, если один поток ожидает ввода с клавиатуры, это может остановить исполнение всех остальных потоков. Для этого требуется реализовать неблокирующие системные вызовы.
2. Ошибка отсутствия страниц - компьютер может иметь такую настройку, что одновременно в оперативной памяти находятся не все программы. Если программа вызывает инструкции (или переходит к инструкциям), отсутствующие в памяти, возникает ошибка обращения к соответствующей странице и операционная система обращается к диску и получает отсутствующие инструкции (и их соседей). Это называется ошибкой отсутствующей страницы. Процесс блокируется до тех пор, пока не будет найдена и считана необходимая инструкция. Если ошибка обращения к отсутствующей странице возникает при выполнении потока, ядро, которое даже не знает о существовании потоков, как и следовало ожидать, блокирует весь процесс до тех пор, пока не завершится дисковая операция ввода-вывода, даже если другие потоки будут готовы к выполнению.
3. Выполнение только одного потока внутри процесса одновременно. Необходимо понимать различие между терминам «parallel» и «concurrent». Parallel – обозначает истинный параллелизм, одному программному потоку соответствует один аппаратный. Concurrent - задачи разбиваются на примерно равные подзадачи для равномерного использования ресурсов, при этом на один аппаратный поток может приходиться более одной подзадачи, что соответствует модели реализации потоков в пользовательском пространстве.

Реализация потоков в ядре.

При данном методе реализации потоков у ядра имеется не только таблица процессов, но и таблица потоков. Пример такой модели изображён на рисунке 2.

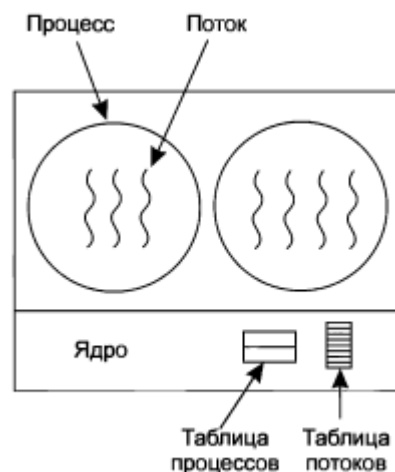


Рисунок 2 – Реализация потоков в ядре.

Для создания или завершения потока необходимо производить системный вызов, блокирующий ядро, что увеличивает накладные расходы.

В некоторых системах, вместо уничтожения старого потока и создания нового, потоки используются повторно. При завершении потока он не удаляется из таблицы потоков в ядре, а лишь отмечается как неспособный к выполнению. При получении запроса на создание нового потока ядро просто активирует один из отмеченных потоков, что заметно снижает накладные расходы.

Данная модель устраняет следующие проблемы модели реализации потоков в пользовательском пространстве:

1. Не требуется создания специальных неблокирующих системных вызовов.
2. Решена проблема ошибки отсутствия страниц. Если один из выполняемых потоков столкнётся с ошибкой обращения к несуществующей странице, ядро может с лёгкостью проверить наличие у процесса любых других готовых к выполнению потоков, и, при наличии таковых, запустить один из них на выполнение, пока будет длиться ожидание извлечения запрошенной страницы диска.

Гибридная реализация.

В этой модели используется планирование потоков как на уровне ядра, так и на уровне пользователя: каждый поток уровня ядра расщепляется на несколько потоков уровня пользователя. На рисунке 3 представлен пример такой модели.

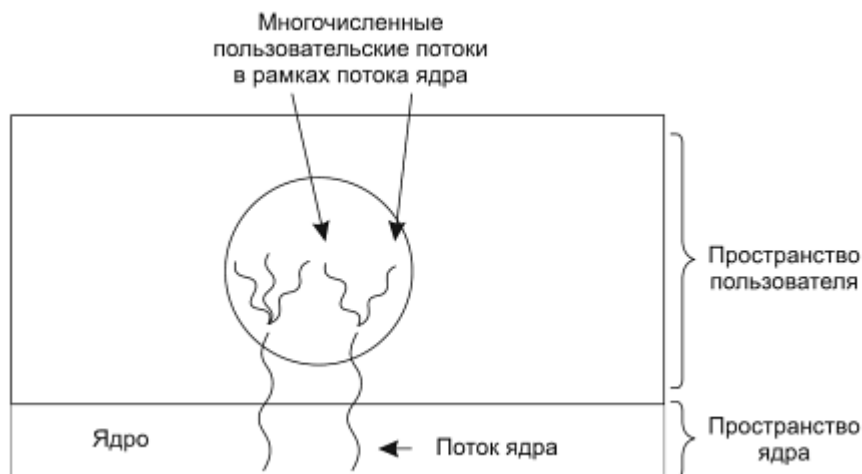


Рисунок 3 – Гибридная реализация потоков.

Данная модель отличается наибольшей гибкостью.

Процессы и потоки в Linux.

Данный раздел содержит как теоретический, так и практический материал. Практический материал частично основан на материале предыдущих лабораторных работ.

Процесс может находиться в следующих состояниях:

- Активный (выполняющийся).
- Ожидающий – процесс в пассивном состоянии, ожидает наступления какого-то события.
- Готовый – процесс готов к исполнению.

Дополнительно выделяют ещё 2 состояния:

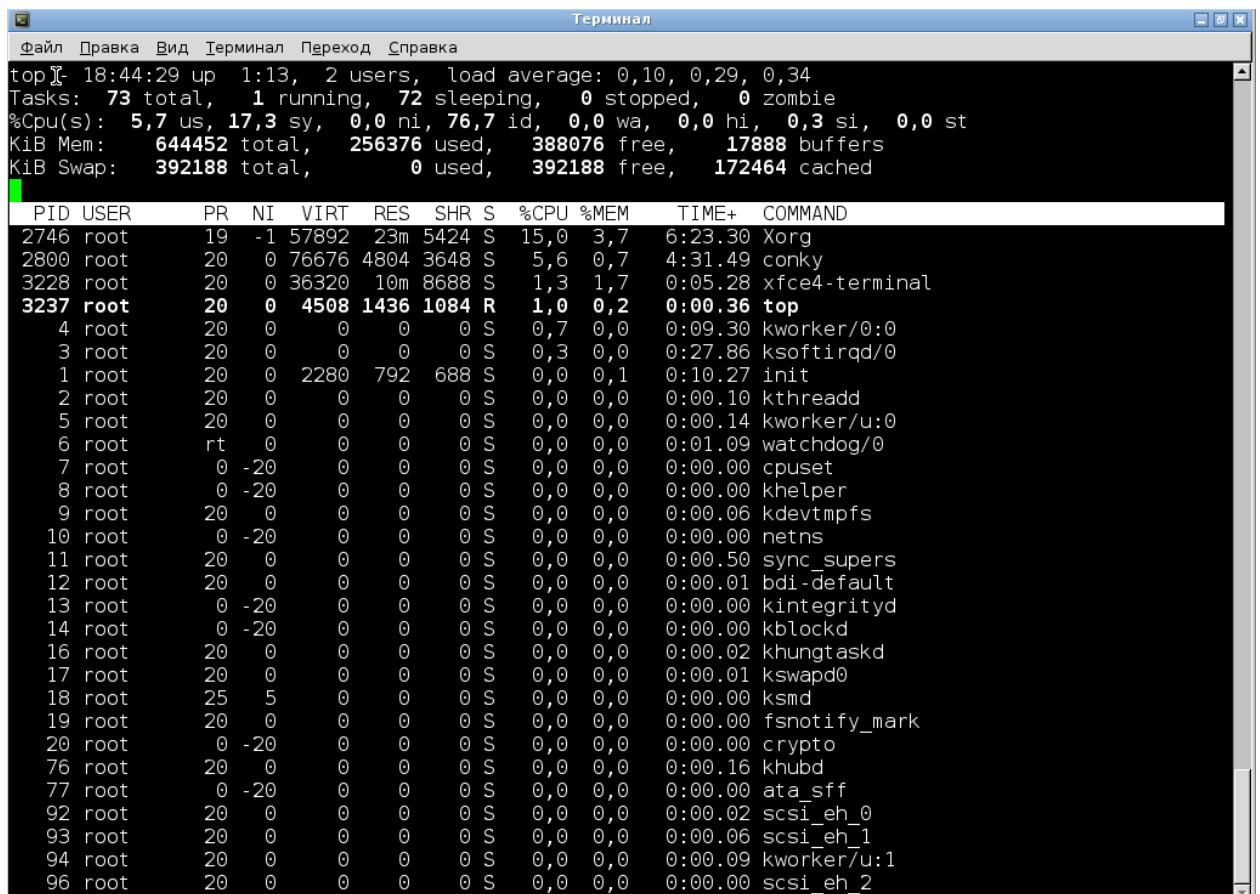
- Рождение (создание) – пассивное состояние, процесс ещё не существует, но существует структура данных для него.
- Смерть (завершение) – процесс завершился, но соответствующая ему структура данных осталась. Такой процесс называется «зомби». Процесс может стать зомби, если он завершился, но его код возврата не был считан родителем.

Команды управления процессами в linux.

Утилита «top»

Начнём с утилиты **top**. Она способна отображать списки процессов и потоков, использование процессора и памяти, и причём в динамике, обновляя информацию раз в секунду, что удобно для выявления наиболее активных процессов.

Дайте в терминале команду top



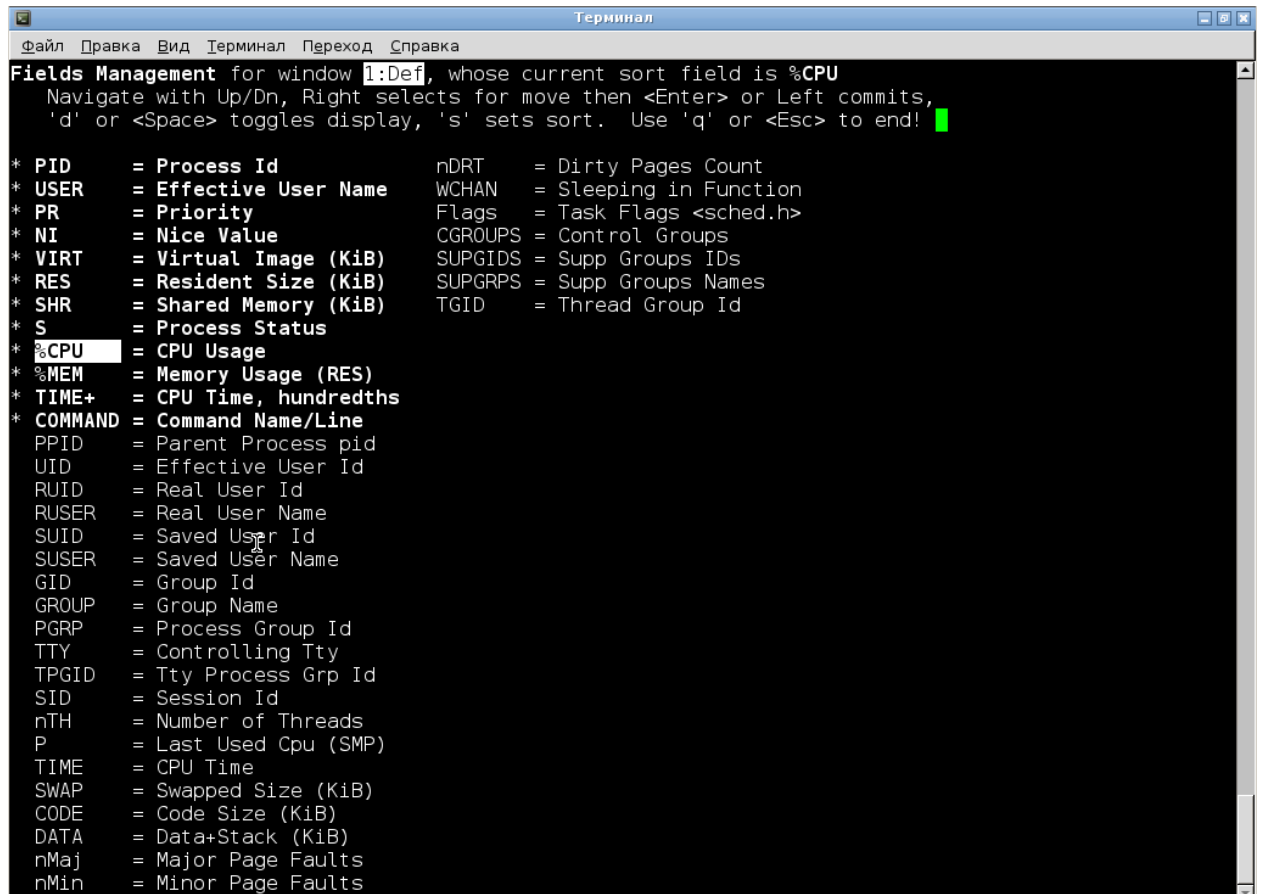
```
Терминал
Файл  Правка  Вид  Терминал  Переход  Справка
top 18:44:29 up 1:13, 2 users, load average: 0,10, 0,29, 0,34
Tasks: 73 total, 1 running, 72 sleeping, 0 stopped, 0 zombie
%Cpu(s): 5,7 us, 17,3 sy, 0,0 ni, 76,7 id, 0,0 wa, 0,0 hi, 0,3 si, 0,0 st
KiB Mem: 644452 total, 256376 used, 388076 free, 17888 buffers
KiB Swap: 392188 total, 0 used, 392188 free, 172464 cached

  PID USER      PR  NI  VIRT  RES  SHR  S  %CPU  %MEM    TIME+  COMMAND
 2746 root        19   -1 57892 23m 5424 S   15,0   3,7   6:23.30 Xorg
 2800 root        20    0 76676 4804 3648 S    5,6   0,7   4:31.49 conky
 3228 root        20    0 36320 10m 8688 S    1,3   1,7   0:05.28 xfce4-terminal
3237 root        20    0 4508 1436 1084 R    1,0   0,2   0:00.36 top
    4 root        20    0      0      0      0 S    0,7   0,0   0:09.30 kworker/0:0
    3 root        20    0      0      0      0 S    0,3   0,0   0:27.86 ksoftirqd/0
    1 root        20    0 2280  792  688 S    0,0   0,1   0:10.27 init
    2 root        20    0      0      0      0 S    0,0   0,0   0:00.10 kthreadd
    5 root        20    0      0      0      0 S    0,0   0,0   0:00.14 kworker/u:0
    6 root        rt    0      0      0      0 S    0,0   0,0   0:01.09 watchdog/0
    7 root         0  -20      0      0      0 S    0,0   0,0   0:00.00 cpuset
    8 root         0  -20      0      0      0 S    0,0   0,0   0:00.00 khelper
    9 root        20    0      0      0      0 S    0,0   0,0   0:00.06 kdevtmpfs
   10 root         0  -20      0      0      0 S    0,0   0,0   0:00.00 netns
   11 root        20    0      0      0      0 S    0,0   0,0   0:00.50 sync_supers
   12 root        20    0      0      0      0 S    0,0   0,0   0:00.01 bdi-default
   13 root         0  -20      0      0      0 S    0,0   0,0   0:00.00 kintegrityd
   14 root         0  -20      0      0      0 S    0,0   0,0   0:00.00 kblockd
   16 root        20    0      0      0      0 S    0,0   0,0   0:00.02 khungtaskd
   17 root        20    0      0      0      0 S    0,0   0,0   0:00.01 kswapd0
   18 root        25    5      0      0      0 S    0,0   0,0   0:00.00 ksm
   19 root        20    0      0      0      0 S    0,0   0,0   0:00.00 fsnotify_mark
   20 root         0  -20      0      0      0 S    0,0   0,0   0:00.00 crypto
   76 root        20    0      0      0      0 S    0,0   0,0   0:00.16 khubd
   77 root         0  -20      0      0      0 S    0,0   0,0   0:00.00 ata_sff
   92 root        20    0      0      0      0 S    0,0   0,0   0:00.02 scsi_eh_0
   93 root        20    0      0      0      0 S    0,0   0,0   0:00.06 scsi_eh_1
   94 root        20    0      0      0      0 S    0,0   0,0   0:00.09 kworker/u:1
   96 root        20    0      0      0      0 S    0,0   0,0   0:00.00 scsi_eh_2
```

Как видно, по умолчанию top отображает процессы (tasks). Нажатием кнопки «Н» можно перейти в режим отображения потоков

```
top - 18:46:53 up 1:16, 2 users, load average: 0,12, 0,22, 0,31
Threads: 153 total, 1 running, 152 sleeping, 0 stopped, 0 zombie
```

Можно изменить состав и порядок колонок. Для этого нужно нажать клавишу «F» во время работы утилиты.



Для того, чтобы отображать количество потоков и группу и ID группы потоков необходимо сделать следующее:

1. Клавишей со стрелкой «вниз» довести курсор до записи nTH.
2. Нажать клавишу со стрелкой «вправо».
3. Нажимая клавишу со стрелкой «вверх», перетащить запись вверх, расположив её после NI.
4. Нажать Enter.
5. Нажать пробел или клавишу «d». После выполнения этого шага nTH должно быть выделено жирным белым шрифтом, а сбоку должна появиться звёздочка.
6. Прodelать аналогичные операции для записи TGID (находится в самом конце списка). После выполнения этих шагов файл настроек должен выглядеть так, как показано на рисунке:

```

Fields Management for window 1:Def, whose current sort field is %CPU
Navigate with Up/Dn, Right selects for move then <Enter> or Left commits,
'd' or <Space> toggles display, 's' sets sort. Use 'q' or <Esc> to end!

* PID      = Process Id          nMin    = Minor Page Faults
* USER     = Effective User Name nDRT    = Dirty Pages Count
* PR       = Priority            WCHAN   = Sleeping in Function
* NI       = Nice Value         Flags    = Task Flags <sched.h>
* nTH      = Number of Threads  CGROUPS = Control Groups
* TGID     = Thread Group Id    SUPGIDS = Supp Groups IDs
* VIRT     = Virtual Image (KiB) SUPGRPS = Supp Groups Names
* RES      = Resident Size (KiB)
* SHR      = Shared Memory (KiB)
* S        = Process Status
* %CPU     = CPU Usage
* %MEM     = Memory Usage (RES)
* TIME+    = CPU Time, hundredths
* COMMAND  = Command Name/Line

```

7. Закрыть настройки (клавишей «q» или Esc).
8. Убедиться, что top работает в режиме отображения потоков (во второй строке должно быть написано «Threads», если там написано «Tasks», нажать клавишу «H»). На рисунке показан результат правильного выполнения данной инструкции.

```

top - 19:06:47 up 1:35, 2 users, load average: 0,09, 0,27, 0,26
Threads: 154 total, 1 running, 153 sleeping, 0 stopped, 0 zombie
%Cpu(s): 3,9 us, 18,9 sy, 0,0 ni, 77,2 id, 0,0 wa, 0,0 hi, 0,0 si, 0,0 st
KiB Mem: 644452 total, 257128 used, 387324 free, 18100 buffers
KiB Swap: 392188 total, 0 used, 392188 free, 172600 cached

```

PID	USER	PR	NI	nTH	TGID	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
2746	root	19	-1	1	2746	58588	23m	5552	S	12,3	3,8	7:53.74	Xorg
2808	root	20	0	8	2800	76676	4812	3648	S	3,6	0,7	2:45.21	conky
3237	root	20	0	1	3237	4624	1512	1096	R	2,6	0,2	0:07.97	top
3228	root	20	0	2	3228	36704	11m	8964	S	1,9	1,8	0:11.45	xfce4-terminal
2800	root	20	0	8	2800	76676	4812	3648	S	1,0	0,7	2:20.18	conky
3241	root	20	0	1	3241	0	0	0	S	0,6	0,0	0:00.50	kworker/0:2
3	root	20	0	1	3	0	0	0	S	0,3	0,0	0:33.84	ksoftirqd/0
2806	root	20	0	8	2800	76676	4812	3648	S	0,3	0,7	0:08.92	conky
2807	root	20	0	8	2800	76676	4812	3648	S	0,3	0,7	0:04.89	conky
1	root	20	0	1	1	2280	792	688	S	0,0	0,1	0:11.12	init
1	root	20	0	1	1	2280	/92	688	S	0,0	0,1	0:11.12	init
2	root	20	0	1	2	0	0	0	S	0,0	0,0	0:00.10	kthreadd
4	root	20	0	1	4	0	0	0	S	0,0	0,0	0:17.38	kworker/0:0
5	root	20	0	1	5	0	0	0	S	0,0	0,0	0:00.14	kworker/u:0
6	root	rt	0	1	6	0	0	0	S	0,0	0,0	0:01.53	watchdog/0
7	root	0	-20	1	7	0	0	0	S	0,0	0,0	0:00.00	cpuset
8	root	0	-20	1	8	0	0	0	S	0,0	0,0	0:00.00	khelper
9	root	20	0	1	9	0	0	0	S	0,0	0,0	0:00.06	kdevtmpfs
10	root	0	-20	1	10	0	0	0	S	0,0	0,0	0:00.00	netns
11	root	20	0	1	11	0	0	0	S	0,0	0,0	0:00.69	sync_supers
12	root	20	0	1	12	0	0	0	S	0,0	0,0	0:00.02	bdi-default
13	root	0	-20	1	13	0	0	0	S	0,0	0,0	0:00.00	kintegrityd
14	root	0	-20	1	14	0	0	0	S	0,0	0,0	0:00.00	kblockd
16	root	20	0	1	16	0	0	0	S	0,0	0,0	0:00.03	khungtaskd
17	root	20	0	1	17	0	0	0	S	0,0	0,0	0:00.01	kswapd0
18	root	25	5	1	18	0	0	0	S	0,0	0,0	0:00.00	ksmd
19	root	20	0	1	19	0	0	0	S	0,0	0,0	0:00.00	fsnotify_mark
20	root	0	-20	1	20	0	0	0	S	0,0	0,0	0:00.00	crypto
76	root	20	0	1	76	0	0	0	S	0,0	0,0	0:00.16	khudb

Для выхода из утилиты top нажмите клавишу «q».

Утилиты «ps» и «kill»

Команда «top» не всегда удобна. Для примера запустим пару процессов:

Дайте команду «w3m www.yandex.ru» и нажмите Ctrl+Z, чтобы перевести процесс в фоновый режим.

Дайте команду «w3m www.yandex.ru» и нажмите Ctrl+Z, чтобы перевести процесс в фоновый режим.

```
[1]+  Stopped                  w3m www.google.ru
root@debian:~# w3m www.yandex.ru

[2]+  Stopped                  w3m www.yandex.ru
root@debian:~# ps
  PID TTY          TIME CMD
  760 pts/0        00:00:00 bash
  766 pts/0        00:00:00 w3m
  774 pts/0        00:00:00 w3m
  789 pts/0        00:00:00 ps
```

Предположим, мы хотим прервать процесс текстового браузера в котором открыт «www.yandex.ru». Знакомая нам команда killall в этом случае бесполезна, поскольку у нас 2 процесса с одинаковым именем «w3m». В этом случае нам поможет команда «kill», но в этой команде надо указать идентификатор процесса. Идентификатор процесса выдаёт команда «top», но оба процесса приостановлены, значит окажутся где то в конце списка среди таких же «спящих процессов» и имя процесса у них одинаковое.

Воспользуемся командой «ps». Эта команда выводит информацию о процессах и завершается, не переходя в интерактивный режим. Из её вывода легко можно отфильтровать нужные строки. Эту команду часто используют для поиска процесса не только по имени, но и по параметрам команды, которой запущен процесс.

Команда ps без параметров выдаёт список процессов, запущенных с текущего терминала.

```
root@debian:~# ps
  PID TTY          TIME CMD
  760 pts/0        00:00:00 bash
  766 pts/0        00:00:00 w3m
  774 pts/0        00:00:00 w3m
  789 pts/0        00:00:00 ps
```

Для нашей задаче информации недостаточно.

Выполним команду «ps a»

```
root@debian:~# ps a
  PID TTY          STAT TIME COMMAND
  464 tty1        Ss   0:00 /bin/login --
  690 tty1        S    0:00 -bash
  697 tty1        S+   0:00 /bin/sh /usr/bin/startx
  719 tty1        S+   0:00 xinit /etc/X11/xinit/xinitrc -- /etc/X11/xinit/xse
  720 tty1        S<   0:04 /usr/bin/X -nolisten tcp :0 vt1 -auth /tmp/servera
  725 tty1        S    0:00 /usr/bin/openbox --startup /usr/lib/i386-linux-gnu
  745 tty1        S    0:00 /usr/bin/dbus-launch --exit-with-session x-session
  749 tty1        S    0:00 /bin/sh /usr/lib/i386-linux-gnu/openbox-autostart
  754 tty1        S    0:00 sh /etc/xdg/openbox/autostart
  755 tty1        S    0:00 idesk
  756 tty1        Sl   0:01 xfce4-terminal
  759 tty1        S    0:00 gnome-pty-helper
  760 pts/0        Ss   0:00 bash
  766 pts/0        T    0:00 w3m www.google.ru
  774 pts/0        T    0:00 w3m www.yandex.ru
  802 pts/0        R+   0:00 ps a
```

Теперь видно, что страница «www.yandex.ru» открыта в процессе PID=774

Попробуем его «убить» командой «**kill 774**», и снова дадим команду «ps a»

```
root@debian:~# ps a
  PID TTY          STAT       TIME COMMAND
  464 tty1      Ss        0:00 /bin/login --
  690 tty1      S         0:00 -bash
  697 tty1      S+        0:00 /bin/sh /usr/bin/startx
  719 tty1      S+        0:00 xinit /etc/X11/xinit/xinitrc -- /etc/X11/xinit/xse
  720 tty1      S<        0:04 /usr/bin/X -nolisten tcp :0 vt1 -auth /tmp/servera
  725 tty1      S         0:00 /usr/bin/openbox --startup /usr/lib/i386-linux-gnu
  745 tty1      S         0:00 /usr/bin/dbus-launch --exit-with-session x-session
  749 tty1      S         0:00 /bin/sh /usr/lib/i386-linux-gnu/openbox-autostart
  754 tty1      S         0:00 sh /etc/xdg/openbox/autostart
  755 tty1      S         0:00 idesk
  756 tty1      Sl        0:01 xfce4-terminal
  759 tty1      S         0:00 gnome-pty-helper
  760 pts/0     Ss        0:00 bash
  766 pts/0     T         0:00 w3m www.google.ru
  774 pts/0     T         0:00 w3m www.yandex.ru
  802 pts/0     R+       0:00 ps a
```

Процесс «774» всё ещё жив. Дело в том, что команда kill не убивает процесс, она посылает ему сигнал «SIGTERM», но не все процессы на него реагируют. Не прореагирует на этот сигнал и зависший процесс. Тем не менее, сначала следует попробовать завершить процесс «по хорошему». Web сервер, например, получив этот сигнал прекратит принимать новые запросы, но завершит выполнять уже поступившие.

Если процесс не завершается, как в нашем случае, будем действовать жёстко.

Дадим команду «**kill -9 774**»

Сигнал SIGKILL(9) прерывает любой процесс кроме «Зомби» процессов. Но помните, дочерние процессы останутся в памяти, временные файлы не удалены, сетевые соединения не закрыты. Подождите секунд 30 после обычного kill, только после этого добивайте.

В документации на конкретные программы можно найти полезные сигналы, которые они обрабатывают. Например, сигнал 1(HUP) просит inetd перезагрузить файл конфигурации.

Вернёмся к выводу команды «ps a»

```
root@debian:~# ps a
  PID TTY          STAT       TIME COMMAND
  464 tty1      Ss        0:00 /bin/login --
  690 tty1      S         0:00 -bash
  697 tty1      S+        0:00 /bin/sh /usr/bin/startx
  719 tty1      S+        0:00 xinit /etc/X11/xinit/xinitrc -- /etc/X11/xinit/xse
  720 tty1      S<        0:04 /usr/bin/X -nolisten tcp :0 vt1 -auth /tmp/servera
  725 tty1      S         0:00 /usr/bin/openbox --startup /usr/lib/i386-linux-gnu
  745 tty1      S         0:00 /usr/bin/dbus-launch --exit-with-session x-session
  749 tty1      S         0:00 /bin/sh /usr/lib/i386-linux-gnu/openbox-autostart
  754 tty1      S         0:00 sh /etc/xdg/openbox/autostart
  755 tty1      S         0:00 idesk
  756 tty1      Sl        0:01 xfce4-terminal
  759 tty1      S         0:00 gnome-pty-helper
  760 pts/0     Ss        0:00 bash
  766 pts/0     T         0:00 w3m www.google.ru
  774 pts/0     T         0:00 w3m www.yandex.ru
  802 pts/0     R+       0:00 ps a
```

В столбце STAT используются следующие обозначения (и их комбинации):

- R: процесс активен.
- S: процесс ожидает (спит менее 20 секунд).
- I: процесс бездействует (спит более 20 секунд).
- D: процесс ожидает ввода-вывода (или другого недолгого события), прерываемый.
- Z: зомби.
- T: процесс остановлен.
- W: процесс выгружен на диск (swap).
- < : процесс в приоритетном режиме.
- N: процесс в режиме низкого приоритета
- L: real-time процесс, имеются страницы, заблокированные в памяти.
- s: лидер сессии, если он завершается, то завершаются и все остальные процессы этой сессии. Примером может служить поведение эмулятора терминала в предыдущей ЛР, когда с помощью него была вызвана панель: если завершился родитель-лидер, то завершались и все процессы этой сессии.
- l: многопоточный процесс.
- + : видимый процесс, т.е. находящийся в группе foreground.

Команда ps также позволяет просмотреть и информацию о запущенных потоках.

```

Терминал
Файл  Правка  Вид  Терминал  Переход  Справка
root@debian:~# ps -elf
UID      PID  PPID  LWP  C  NLWP  STIME  TTY      TIME  CMD
root      1    0     1  0    1  17:30  ?        00:00:09  init [2]
root      2    0     2  0    1  17:30  ?        00:00:00  [kthreadd]
root      3    2     3  0    1  17:30  ?        00:00:20  [ksoftirqd/0]
root      4    2     4  0    1  17:30  ?        00:00:05  [kworker/0:0]
root      5    2     5  0    1  17:30  ?        00:00:00  [kworker/u:0]
root      6    2     6  0    1  17:30  ?        00:00:00  [watchdog/0]
root      7    2     7  0    1  17:30  ?        00:00:00  [cpuset]
root      8    2     8  0    1  17:30  ?        00:00:00  [khelper]
root      9    2     9  0    1  17:30  ?        00:00:00  [kdevtmpfs]
root     10    2    10  0    1  17:30  ?        00:00:00  [netns]
root     11    2    11  0    1  17:30  ?        00:00:00  [sync_supers]
root     12    2    12  0    1  17:30  ?        00:00:00  [bdi-default]
root     13    2    13  0    1  17:30  ?        00:00:00  [kintegrityd]
root     14    2    14  0    1  17:30  ?        00:00:00  [kblockd]
root     16    2    16  0    1  17:30  ?        00:00:00  [khungtaskd]
root     17    2    17  0    1  17:30  ?        00:00:00  [kswapd0]
root     18    2    18  0    1  17:30  ?        00:00:00  [ksmd]
root     19    2    19  0    1  17:30  ?        00:00:00  [fsnotify_mark]
root     20    2    20  0    1  17:30  ?        00:00:00  [crypto]
root     76    2    76  0    1  17:31  ?        00:00:00  [khubd]
root     77    2    77  0    1  17:31  ?        00:00:00  [ata_sff]
root     92    2    92  0    1  17:31  ?        00:00:00  [scsi_eh_0]
  
```

Как видно из рисунка, появились два новых поля:

- LWP – ID потока (Light Weight Process).
- NLWP – количество потоков в процессе (Number of LWPs).

Объединение команд

Конвейеры

Часто требуется прервать невидимый системный процесс (в linux такие процессы называются демонами, поэтому у многих процессов имя оканчивается на «d»).

Команда «**ps axu**» выдаёт полный список процессов. Но теперь их слишком много. Все на экране не умещаются.

Чтобы просмотреть список процессов «постранично» дайте команду «**ps axu | more**»

Для перехода к следующей странице нажмите пробел. Для поиска текста «google» наберите «/google» и нажмите ENTER. Для выхода нажмите q.

Командой «ps axu | more» мы запускаем 2 процесса в **конвейере**.

Вывод команды «ps axu» направляется на вход команды more, которая осуществляет постраничный вывод.

Постраничный вывод освоили, теперь можно установить другой эмулятор терминала из xfce 4, который поддерживает скроллинг.

apt-get install xfce4-terminal

Особенно удобно объединять команду «ps axu» в конвейер с утилитой grep,

Дайте команду «ps axu | **grep google**»

```
root@debian:~# ps axu | grep google
root      778  0.0  0.8  9340  5504 pts/0    T   21:57   0:00 w3m www.google.ru
root      946  0.0  0.3  4560  2348 pts/0    S+  23:37   0:00 grep google
```

Утилита «grep» отбирает из входного потока строки, содержащие текст заданный первым параметром. Если в искомой строке содержатся пробелы, обязательно заключите такую строку в апострофы «ps axu | grep '**w3m www.google.ru**'», иначе «grep» будет искать строку «w3m» в файле «www.google.ru». Пробел разделяет в команде параметры.

Утилита grep поддерживает «регулярные выражения». Символы «.<>[]|\» служебные. В частности, в квадратных скобках задаётся диапазон символов которые могут стоять в этой позиции строки. Например? [a-z]

Выполните команду «ps axu | grep '**[]www.google.ru**'» и попробуйте догадаться, как она работает и какая от неё польза.

Кроме конвейеров существует ещё несколько способов объединения команд.

Параллельное выполнение команд

Команда1 & команда 2 — запускает команду 1 в фоновом режиме и сразу начинает выполнение команды 2, не дожидаясь завершения работы команды1

В частности команда «команда &» сразу запустит эту команду в фоновом режиме.

Теперь вы знаете, почему в файле «autostart» мы писали «xfce4-panel &»

Запуск команды в фоновом режиме, с указанием «&» отличается от перевода процесса в фоновый режим нажатием Ctrl+Z. По Ctrl+Z процесс получает сигнал SIGTSTR(20) и переходит в состояние «Stoped» (Приостановлен). Чтобы продолжить фоновое выполнение процесса надо дать команду «bg».

Последовательное выполнение команд

Команда1 ; команда 2 — последовательно выполняет команды 1 и команды 2

Например, команда «kill 111 ; sleep 30 ; kill -9 111» попытается остановить процесс 111, затем подождёт 30 секунд, затем принудительно убьёт этот процесс.

Команды, объединённые «;» выполняются все, все, вне зависимости от результатов работы. Не всегда это хорошо.

Команда1 && команда 2 — выполняет команду 1 и в случае успешного завершения выполняет команду 2.

Например, команду отправки архива на другой компьютер следует давать только в случае успешного архивирования нужных файлов.

Команда1 || команда 2 — выполняет команду 1 и в случае ошибочного завершения выполняет команду 2.

Например, команда «ping -c 1 -W 3 192.168.99.99 || mail -s 'сервер недоступен' test@mail.ru» отправит e-mail с текстом 'сервер недоступен', если не отвечает сервер с IP адресом «192.168.99.99». (если у вас установлена и настроена утилита отправки электронной почты mail)

Последовательное выполнение группы команд в фоновом режиме

Выполните команду (sleep 5 && pstree)&

Запускается фоновый процесс, в котором, с задержкой 5секунд выполнится команда pstree..В течении этих 5 секунд вы можете выполнять команды в основном процессе.

```
root@debian:~# (sleep 2;pstree)&
[1] 726
root@debian:~# systemd└─acpid
                        └─atd
                        └─auditd─{auditd}
                        └─cron
                        └─dbus-daemon
                        └─dhclient
                        └─exim4
                        └─login─bash─bash─pstree
```


Обратите внимание на результаты команды «pstree». Команды, заключённые в круглые скобки выполняются в новом экземпляре интерпретатора команд `bash`

Подстановка результатов одной команды в другую

Пусть у нас ежедневно ночью, автоматически запускается процесс архивирования файлов из заданного каталога, и я хочу, чтобы архив создавался каждый день с новым именем, чтобы можно было восстановить файлы, удалённые по ошибке несколько дней назад. Для этого достаточно подставить текущую дату в имя файла.

Начнём с команды «`date`». Эта команда позволяет не только установить системную дату и время, но и вывести его в заданном формате.

Например, команда «`date +%Y-%m-%d`» выводит текущую дату в формате год-месяц-день. Формат год-месяц-день удобен для сортировки файлов. При сортировке по имени файлы окажутся отсортированными по имени.

Команда «`tar -cvzf ~/archive$(date +%Y-%m-%d).tar ~/folder/*`» подставит результаты работы команды «`date`» на место, `$(...)` и выполнит получившуюся команду.

Перенаправление входных и выходных потоков

В стандартной библиотеке языка C и других языков программирования есть константы «`stdin`, `stdout`, `stderr`», типа поток. Именно в `stdout` осуществляют вывод все команды `linux` данные из этого потока принимает эмулятор терминала и выводит на экран. Ввод с клавиатуры попадает в `stdin` и принимается интерпретатором команд `bash`.

Эти потоки можно перенаправить.

При организации рассмотренного выше конвейера, `stdout` первой команды перенаправляется в `stdin` второй команды. При этом, ничего на диск не записывается, если вторая команда обрабатывает данные быстрее, чем первая их поставляет, то её процесс переводится в режим ожидания, пока первый процесс не запишет в выходной поток хотя бы один байт. И наоборот, если второй поток не успевает обрабатывать данные, то как только заполнится небольшой буфер обмена, первый поток перейдёт в режим ожидания, пока буфер не освободится.

Выходной поток можно записать в файл. Для этого надо задать в команде «`>file`»

Например, команда «`date > xx.txt`» запишет вывод команды `date` в файл «`xx.txt`».

Просмотреть файл можно командой «`cat xx.txt`»

```
root@debian:~# date>xx.txt
root@debian:~# cat xx.txt
Пт окт 21 13:42:41 MSK 2016
```

При повторном выполнении команды файл заменяется. Если мы хотим дописать результаты команды в конец существующего файла, надо задать в команде «`>>file`».

Например, так можно организовать ведение файла протокола.

```
date >> log.txt
```

```
echo "Архивация данных...">>log.txt
```

```
tar -zcvf archive.tar.gz ~/.idesktop/ >> log.txt
```

```
root@debian:~# date >> log.txt
root@debian:~# echo "Архивация данных...">>log.txt
root@debian:~# tar -zcvf archive.tar.gz ~/.idesktop/ >> log.txt
tar: Удаляется начальный '/' из имен объектов
root@debian:~# cat log.txt
Пт окт 21 14:08:15 MSK 2016
Архивация данных...
/root/.idesktop/
/root/.idesktop/default.lnk
```

Командой «date» записываем в файл текущее время (при анализе файла протокола мы должны знать когда произошла ошибка)

Командой «echo» записываем в файл строку комментария на производимые действия.

Команда «tar» - архивирует, в нашем примере, каталог .idesktop (эту команду подробно будем рассматривать в следующей лабораторной работе)

Обратите внимание, что несмотря на то, что вывод команды tar был перенаправлен в файл, на экран было выведено предупреждение «**tar: Удаляется ...**»

Дело в том, что этот текст выводится не в stdout, а в stderr, а он не перенаправлен.

Можно перенаправить stderr в другой файл, и тогда у вас будет отдельный протокол ошибок, указав в команде «2>file». Например:

```
«tar -zcvf archive.tar.gz ~/.idesktop/ >>log.txt 2>>err.txt»
```

А можно перенаправить и stdout и stderr в один и тот же файл, и тогда у вас будет общий протокол, указав в команде &>file. Например:

```
«tar -zcvf archive.tar.gz ~/.idesktop/ &>>log.txt»
```

Если вы пишете программу и вызываете в ней дочерний процесс, обязательно направьте весь вывод этого процесса в файл или обеспечьте чтение из stdout и stderr. Иначе буфер одного из этих потоков заполнится и дочерний процесс остановится. В вы в основном процессе вечно будете ждать его завершения.

А ещё можно подавить вывод сообщений об ошибках, используя «2>&-»

Ограничения на ресурсы процесса

В linux имеется возможность ограничить ресурсы, предоставляемые процессу с помощью команды ulimit. Для начала посмотрим действующие ограничения с помощью команды «ulimit -a»

```
root@debian:~# ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority     (-e) 0
file size               (blocks, -f) unlimited
pending signals         (-i) 4904
max locked memory       (kbytes, -l) 64
max memory size         (kbytes, -m) unlimited
open files              (-n) 65536
pipe size               (512 bytes, -p) 8
POSIX message queues    (bytes, -q) 819200
real-time priority      (-r) 0
stack size              (kbytes, -s) 8192
cpu time                (seconds, -t) unlimited
max user processes      (-u) 4904
virtual memory          (kbytes, -v) unlimited
file locks              (-x) unlimited
```

В скобках указаны опции, которые управляют этими параметрами.

Особый интерес представляют ограничения на выделяемую память (-m) и на время выполнения (-t).

Но прежде чем выполнять из командной строки эту команду, запомните, что это ограничение будет действовать на все процессы, запускаемые из интерпретатора команд (bash), связанного с текущим терминалом. Поэтому, надо выполнить команды в дочернем bash.

В разделе «Последовательное выполнение группы команд в фоновом режиме» мы пользовались для этого круглыми скобками. Попробуем альтернативный способ.

Команда «`bash -c 'команда'`» запускает дочерний интерпретатор команд и выполняет в нём заданную команду. Обратите внимание, я заключил команду в прямые одиночные кавычки. В отличие от двойных кавычек, Linux передает такую строку запускаемой программе как единый параметр, не заглядывая в него, благодаря этому в команде можно использовать символы объединения потоков, не опасаясь, что текущий интерпретатор команд начнёт их интерпретировать, как свои.

Для примера, запустим команду `apt-get update` с ограничением по времени выполнения: «`bash -c 'ulimit -t 2 && apt-get update'`»

```

root@debian:~# bash -c 'ulimit -t 2 && apt-get update'
Игн http://ftp.debian.org jessie InRelease
В кэшe http://security.debian.org jessie/updates InRelease
В кэшe http://ftp.debian.org jessie Release.gpg
В кэшe http://ftp.debian.org jessie Release
В кэшe http://security.debian.org jessie/updates/main Sources
В кэшe http://ftp.debian.org jessie/main Sources
В кэшe http://security.debian.org jessie/updates/main i386 Packages
В кэшe http://ftp.debian.org jessie/main i386 Packages
В кэшe http://security.debian.org jessie/updates/main Translation-en
В кэшe http://ftp.debian.org jessie/main Translation-ru
В кэшe http://ftp.debian.org jessie/main Translation-en
E: Method copy has died unexpectedly!
E: Порождённый процесс copy получил сигнал 9.

```

Этой командой мы запускаем в дочернем процессе интерпретатор команд `bash` и просим его выполнить команду «`ulimit -t 2 && apt-get update`».

Дочерний интерпретатор команд сначала устанавливает ограничение времени команду «`ulimit`», а затем выполняет команду «`apt-get update`».

Обратите внимание, что «`apt-get update`» прервалась гораздо позже, чем через 2 секунды. Дело в том, что «`-t`» накладывает ограничение на чистое время работы процессора. В это время не входит время дисковых операций. А если процесс перешёл в состояние ожидания (например, ждёт ввода данных от пользователя или соединения по сети), процессорное время вообще не расходуется и прерывания не произойдёт.

Чтобы ограничить полное время выполнения команды, следует использовать команду «`timeout -s сигнал время команда`»

Например, «`timeout -s 9 2 apt-get update`»

Запуск процесса по расписанию

В Linux имеется планировщик задач «`cron`». Каждый пользователь может настроить запуск программ по расписанию и они будут выполняться от его имени в заданное время.

Для того, чтобы настроить расписания следует дать команду «`crontab -e`»

В редакторе «`напо`» открывается конфигурационный файл текущего пользователя. Текстовый редактор «`напо`» имеет привычный интерфейс, и подсказки по командам в нижней части экрана. Возможно, запустится другой текстовый редактор в зависимости от установленных пакетов.

Рассмотрим конфигурационный файл.

Каждая строка (кроме комментариев и пустых строк) определяет день, время и выполняемую команду, которые задаются через пробел в следующей последовательности:

Последовательность	Минуты	Часы	Дни месяца	Месяцы	Дни недели	Команда
Диапазон	0-59,*	0-23,*	1-31,*	1-12,*	1-7,*	

Примеры значений	0 * */10	0 */12	*	*	* 1-5 6,7	
Пример1	*/10	*	*	*	*	date>>log1.txt

Значения могут задаваться в виде:

- Диапазона. Например, 1-5 Означает множество чисел {1,2,3,4,5}
- Перечисления. Например, {6,7} Означает множество чисел {6,7}
- Кратного. Например для часа */2 означает раз в 2 часа
- Любое значение (*).

ВНИМАНИЕ! После последней строки должен быть перевод строки, иначе cron не будет выполнять эту команду

Примеры:

Выполнять по выходным в 01:00:

0 1 * * 6,7 Команда

Выполнять только по будням в 01:30

30 1 * * 1-5 Команда

Выполнять каждый час в 0 минут

0 * * * * Команда

Выполнять каждые 10 минут

*/10 * * * * Команда

Другие команды управления процессами.

pidof – утилита, выдающая pid процесса по его имени.

kill – завершение процесса с определённым pid, возможна отправка сигнала: kill [-номер сигнала] PID.

killall – завершает все процессы с совпадающим именем, в остальном команда похожа на kill.

nice – изменить приоритет процесса для планировщика. По умолчанию понижает приоритет процесса, по умолчанию он равен 10. Позволяет установить приоритет от -20 (наивысший приоритет) до 19 (низший, в FreeBSD низшим является приоритет 20). Для повышения приоритета процесса надо писать смещение приоритета с двойным минусом: --5. Следует отметить, что nice не устанавливает приоритет, а указывает смещение приоритета. Поле nice (NI) есть, например, на рисунке 10.

nohup – игнорирование всех сигналов прерывания, кроме SIGHUP и SIGQUIT.

jobs – показывает список процессов, выполняющихся в фоновом режиме.

bg – продолжить исполнение приостановленной программы в фоновом режиме.

fg – вывод программы в обычный режим (foreground, «на передний план»).

POSIX.

POSIX (Portable Operating System Interface for Unix) – набор стандартов, определяющих взаимодействие ОС с прикладными программами. Разрабатывается IEEE (Institute of Electrical and Electronical Engineers), комитетом 1003. Состоит из следующих частей:

1. POSIX 1003.1 – определяет стандарт на основные компоненты ОС, API для процессов, ФС и т.д.
2. POSIX 1003.1b – стандарт на расширения «реального времени» (планировка приоритетов, семафоры, передача сообщений, сигналы реального времени, асинхронный и синхронный ввод/вывод, разделяемая память и т.д.).
3. POSIX 1003.1c – стандарт на потоки (создание и завершение, планирование, синхронизация, обработка сигналов).
4. POSIX 1003.1d – стандарт на дополнительные расширения «реального времени» (обработчики прерываний и т.д.).
5. POSIX 1003.2 – стандарт на основные утилиты.

Это далеко не все стандарты POSIX. Описанные выше версии стандарта являются достаточно старыми (1991-1992 годы), однако их хватает для большинства задач. Помимо POSIX существуют и другие стандарты.

ОС, полностью отвечающие стандартам POSIX, называются POSIX-совместимыми, или полностью совместимыми. Также существуют частично совместимые и по большей части совместимые. Для официального признания совместимости ОС должна пройти сертификацию, потому некоторые POSIX-совместимые ОС могут находиться в группе «частично совместимых».

В Приложении 2 описывается работа с потоками POSIX (POSIX Threads, Pthreads).

D-Bus.

D-Bus представляет собой систему межпроцессного взаимодействия (IPC, Inter-Process Communication). Эта система предоставляет различные шины для обмена сообщениями. Система была разработана в рамках проекта freedesktop.org для возможности коммуникации процессов вне зависимости от окружения рабочего стола.

В данной лабораторной работе будут рассмотрены сообщения, передаваемые по различным шинам. Для этого будет использована утилита dbus-monitor.

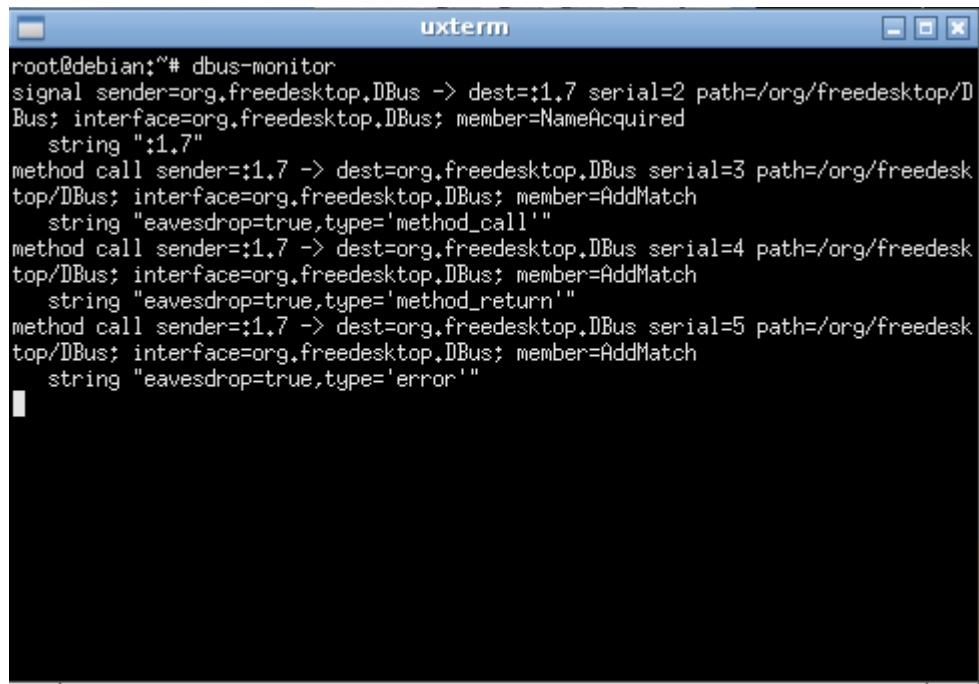
Некоторые опции утилиты dbus-monitor:

--system – мониторинг системной шины сообщений.

--session – мониторинг шины сообщений сессии (по умолчанию).

Для выхода необходимо нажать Ctrl-\ (или Ctrl-C).

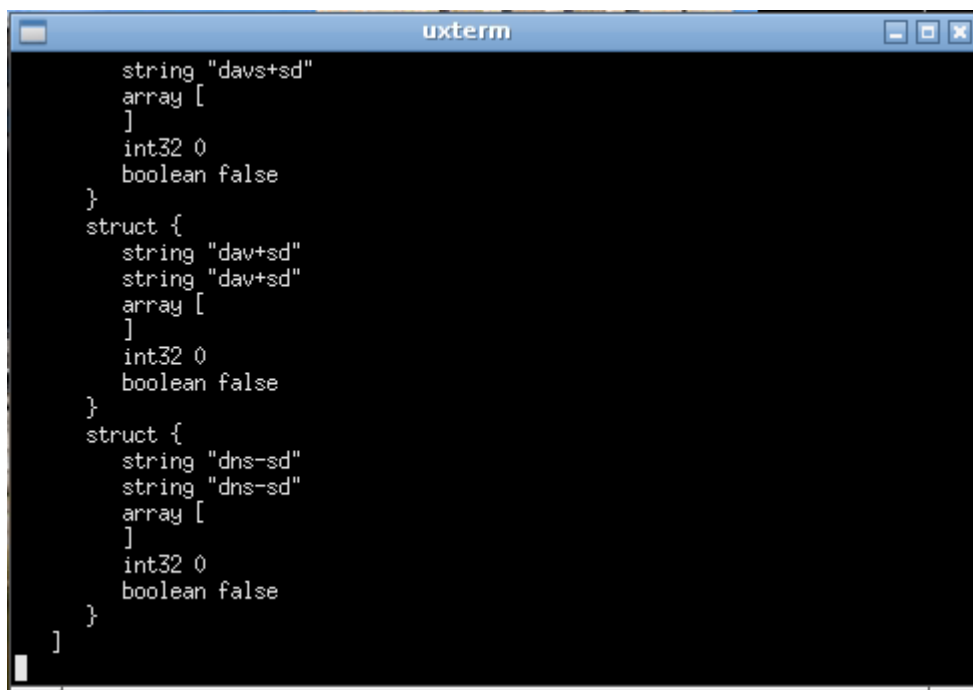
Просмотрим сообщения, переданные по шине сессии (система сразу после запуска).

A terminal window titled 'uxterm' showing the output of the 'dbus-monitor' command. The output displays several DBus messages, including a signal from org.freedesktop.DBus and two method calls to org.freedesktop.DBus.AddMatch, each with an 'eavesdrop=true' string.

```
root@debian:~# dbus-monitor
signal sender=org.freedesktop.DBus -> dest=:1.7 serial=2 path=/org/freedesktop/DBus; interface=org.freedesktop.DBus; member=NameAcquired
  string ":1.7"
method call sender=:1.7 -> dest=org.freedesktop.DBus serial=3 path=/org/freedesktop/DBus; interface=org.freedesktop.DBus; member=AddMatch
  string "eavesdrop=true,type='method_call'"
method call sender=:1.7 -> dest=org.freedesktop.DBus serial=4 path=/org/freedesktop/DBus; interface=org.freedesktop.DBus; member=AddMatch
  string "eavesdrop=true,type='method_return'"
method call sender=:1.7 -> dest=org.freedesktop.DBus serial=5 path=/org/freedesktop/DBus; interface=org.freedesktop.DBus; member=AddMatch
  string "eavesdrop=true,type='error'"

```

Сообщения не отличаются разнообразием. Не закрывая окно эмулятора терминала с запущенным dbus-monitor, запустите Leafpad через панель xfce. После этого в окне dbus-monitor произойдут изменения, показанные на следующем рисунке.

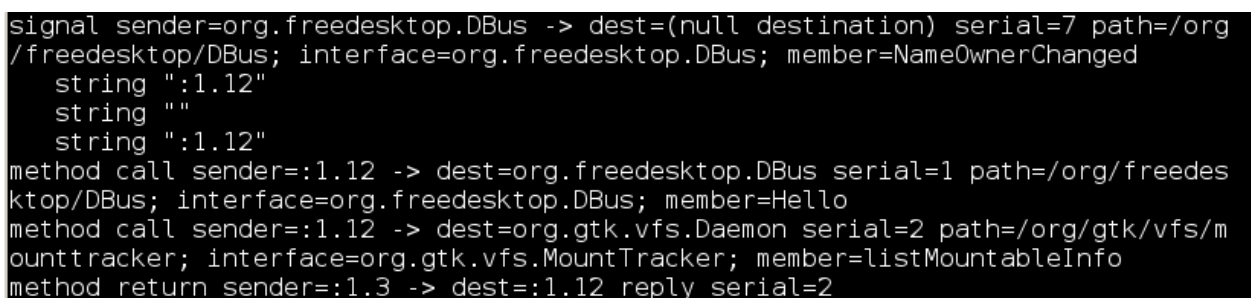
A terminal window titled 'uxterm' showing the output of the 'dbus-monitor' command. The output displays a complex DBus message structure, including a signal from org.freedesktop.DBus and two method calls to org.freedesktop.DBus.AddMatch, each with an 'eavesdrop=true' string.

```

  string "davs+sd"
  array [
  ]
  int32 0
  boolean false
}
struct {
  string "dav+sd"
  string "dav+sd"
  array [
  ]
  int32 0
  boolean false
}
struct {
  string "dns-sd"
  string "dns-sd"
  array [
  ]
  int32 0
  boolean false
}
}
]

```

На следующем рисунке показано начало этого сообщения.

A terminal window showing the output of the 'dbus-monitor' command. The output displays a complex DBus message structure, including a signal from org.freedesktop.DBus and two method calls to org.freedesktop.DBus.AddMatch, each with an 'eavesdrop=true' string.

```
signal sender=org.freedesktop.DBus -> dest=(null destination) serial=7 path=/org/freedesktop/DBus; interface=org.freedesktop.DBus; member=NameOwnerChanged
  string ":1.12"
  string ""
  string ":1.12"
method call sender=:1.12 -> dest=org.freedesktop.DBus serial=1 path=/org/freedesktop/DBus; interface=org.freedesktop.DBus; member=Hello
method call sender=:1.12 -> dest=org.gtk.vfs.Daemon serial=2 path=/org/gtk/vfs/mounttracker; interface=org.gtk.vfs.MountTracker; member=listMountableInfo
method return sender=:1.3 -> dest=:1.12 reply_serial=2

```

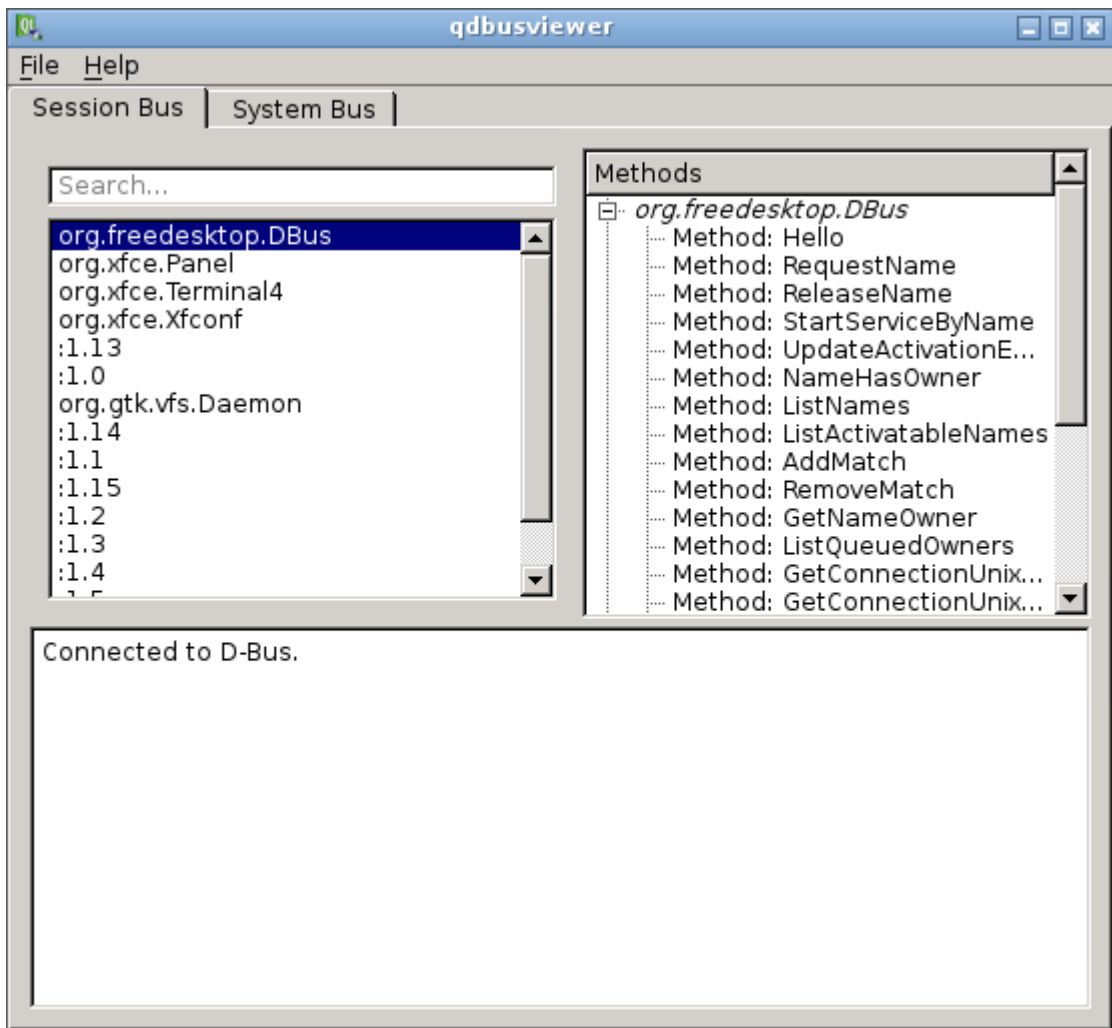
Попытайтесь самостоятельно расшифровать последнее сообщение.

Можно отправлять сообщения по D-Bus прямо из консоли, для этого служит команда `dbus-send`.

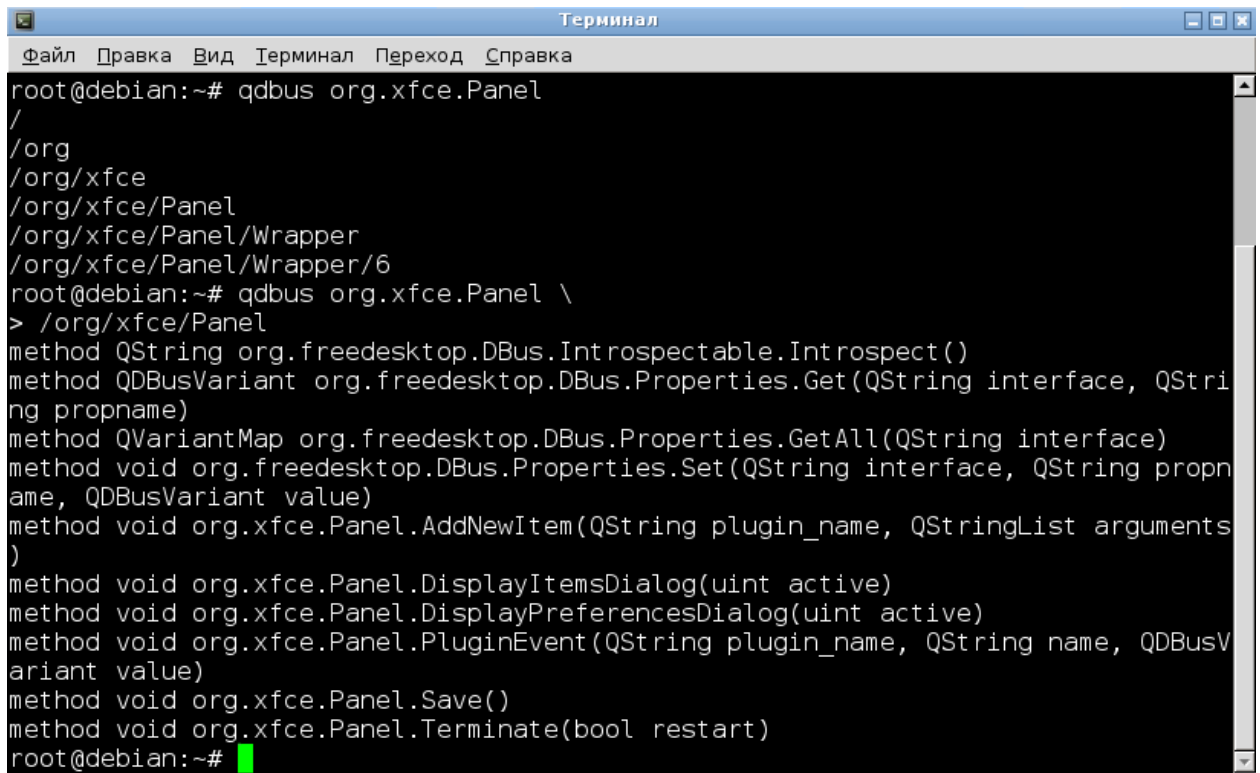
Библиотека Qt поддерживает работу с D-Bus. Для работы с D-Bus используются два приложения: `qdbus` (консольный интерфейс) и `qdbusviewer` (графический интерфейс). Это приложение находится в пакете `qt4-dev-tools`:

```
apt-get install qt4-dev-tools
```

Внимание! Для установки этого пакета требуется скачать достаточно большой объём данных (137 Мб), что может занять достаточно много времени.



Как видно, утилита `qdbusviewer` позволяет просматривать интерфейсы и их методы для приложений, использующих D-Bus. Это же можно сделать с помощью консольной утилиты `qdbus`.



```
Терминал
Файл Правка Вид Терминал Переход Справка
root@debian:~# qdbus org.xfce.Panel
/
/org
/org/xfce
/org/xfce/Panel
/org/xfce/Panel/Wrapper
/org/xfce/Panel/Wrapper/6
root@debian:~# qdbus org.xfce.Panel \
> /org/xfce/Panel
method QString org.freedesktop.DBus.Introspectable.Introspect()
method QDBusVariant org.freedesktop.DBus.Properties.Get(QString interface, QString propName)
method QVariantMap org.freedesktop.DBus.Properties.GetAll(QString interface)
method void org.freedesktop.DBus.Properties.Set(QString interface, QString propName, QDBusVariant value)
method void org.xfce.Panel.AddNewItem(QString plugin_name, QStringList arguments)
method void org.xfce.Panel.DisplayItemsDialog(uint active)
method void org.xfce.Panel.DisplayPreferencesDialog(uint active)
method void org.xfce.Panel.PluginEvent(QString plugin_name, QString name, QDBusVariant value)
method void org.xfce.Panel.Save()
method void org.xfce.Panel.Terminate(bool restart)
root@debian:~#
```

На рисунке просматриваются интерфейсы и методы для объекта `org.xfce.Panel`. С помощью `qdbus` можно вызывать эти методы. Прими, перезапуск панели (набирается в одну строчку):

```
qdbus org.xfce.Panel /org/xfce/Panel
org.xfce.Panel.Terminate true
```

Для удобства команду `qdbus` разбивают по строчкам «объект-интерфейс-метод»:

```
qdbus org.xfce.Panel \
/org/xfce/Panel \
org.xfce.Panel.Terminate true
```

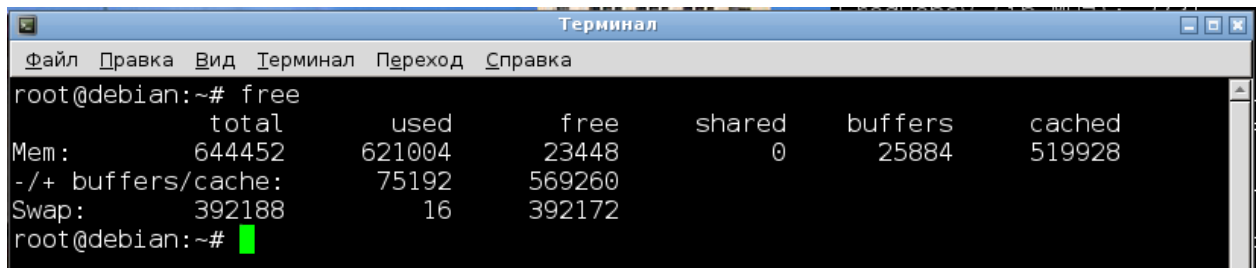
Вместо вызова метода с помощью `qdbus` можно воспользоваться отправкой сообщения:

```
dbus-send --dest=org.xfce.Panel \
/org/xfce/Panel \
org.xfce.Panel.Terminate \
boolean:true
```

Управление оперативной памятью в Linux.

Данная секция описывает некоторые полезные команды для мониторинга состояния ОЗУ. Предполагается, что предварительно был освоен теоретический материал.

Для мониторинга оперативной памяти можно использовать утилиту free. Эта утилита показывает статистику по использованию ОЗУ и файла подкачки (swap). Пример работы показан на рисунке 16.



```
root@debian:~# free
              total        used        free      shared    buffers     cached
Mem:           644452      621004       23448          0       25884     519928
-/+ buffers/cache:       75192     569260
Swap:          392188         16      392172
root@debian:~#
```

Рисунок 16 – Вывод утилиты free

Строка «-/+ buffers/cache» показывает, сколько памяти использовано и сколько свободно с точки зрения её использования в приложениях.

Некоторые ключи, используемые с утилитой free:

-b (--bytes) – показывать память в байтах.

-k (--kilo) – показывать память в килобайтах (используется по умолчанию).

-m (--mega) – показывать память в мегабайтах.

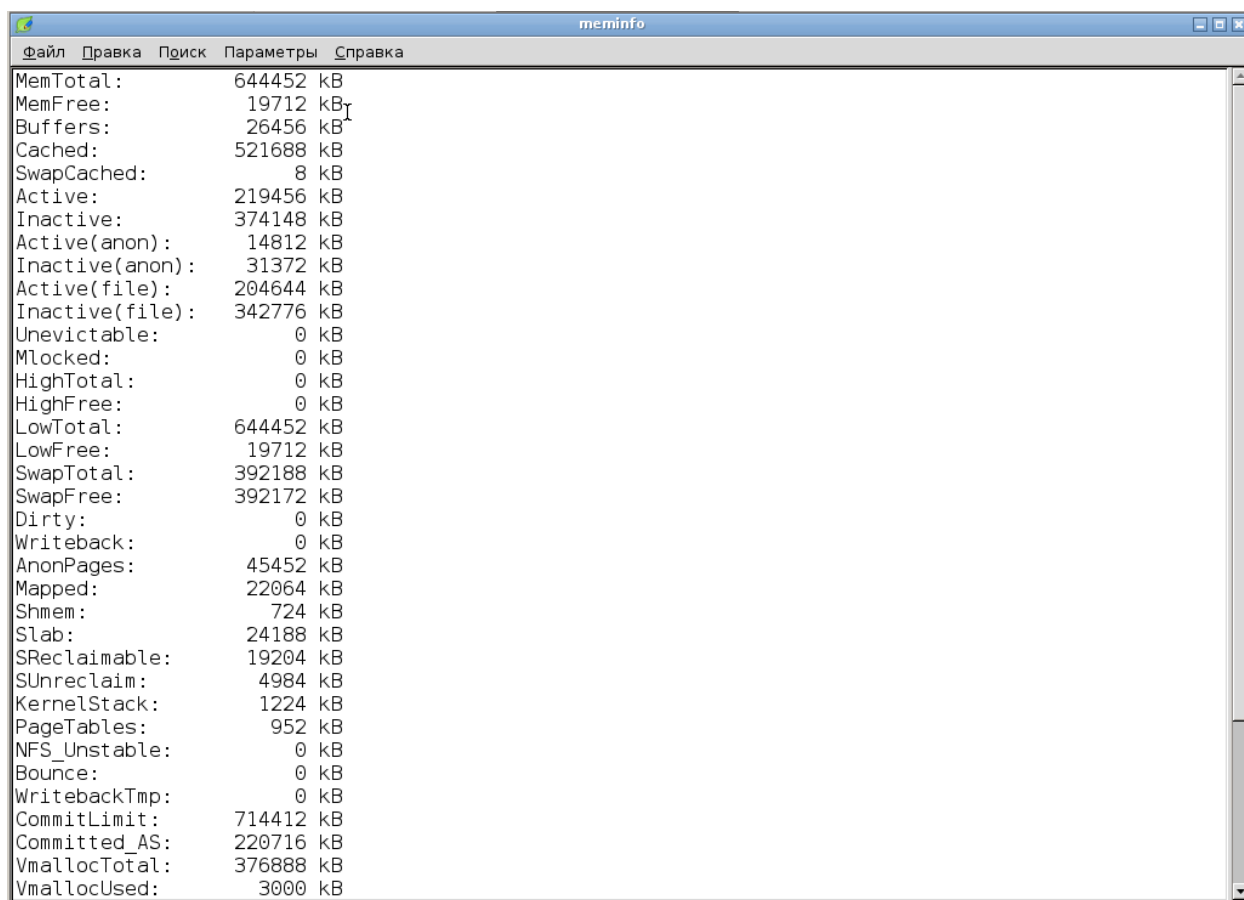
-g (--giga) – показывать память в гигабайтах.

--tera – показывать память в терабайтах.

--si – использовать десятичную систему, т.е. делить не на 1024, а на 1000.

-h (--human) – число в колонке округляется до 3 знаков и к нему прибавляется единица измерения.

Более подробную информацию об использовании памяти можно посмотреть в файле `/proc/meminfo`, именно этот файл использует утилита `free`. На рисунке 17 показан пример такого файла.



MemTotal:	644452 kB
MemFree:	19712 kB
Buffers:	26456 kB
Cached:	521688 kB
SwapCached:	8 kB
Active:	219456 kB
Inactive:	374148 kB
Active(anon):	14812 kB
Inactive(anon):	31372 kB
Active(file):	204644 kB
Inactive(file):	342776 kB
Unevictable:	0 kB
Mlocked:	0 kB
HighTotal:	0 kB
HighFree:	0 kB
LowTotal:	644452 kB
LowFree:	19712 kB
SwapTotal:	392188 kB
SwapFree:	392172 kB
Dirty:	0 kB
Writeback:	0 kB
AnonPages:	45452 kB
Mapped:	22064 kB
Shmem:	724 kB
Slab:	24188 kB
SReclaimable:	19204 kB
SUnreclaim:	4984 kB
KernelStack:	1224 kB
PageTables:	952 kB
NFS_Unstable:	0 kB
Bounce:	0 kB
WritebackTmp:	0 kB
CommitLimit:	714412 kB
Committed_AS:	220716 kB
VmallocTotal:	376888 kB
VmallocUsed:	3000 kB

Попытайтесь самостоятельно расшифровать файл `meminfo`.

Slab – это кэш-память для `dentry` и `inode`, она будет рассмотрена в лабораторной работе, посвящённой файловым системам.

В данной методичке приведены приложения.

Первое приложение может быть использовано практически всеми.

Второе приложение предназначено в основном для программистов, но может быть использовано и другими пользователями.

Третье приложение содержит дополнительную информацию по процессам и потокам в Linux".

Выводы.

Linux имеет гибкие механизмы управления процессами и потоками. Процессы могут посылать друг другу сигналы или сообщения, обмениваться данными через файлы или каналы. Суперпользователь может изменять приоритеты процессов. Также Linux имеет стандартные гибкие средства мониторинга ресурсов.

Выводы по D-Bus и управлению оперативной памятью предлагается сделать самостоятельно.

Порядок выполнения работы.

1. Ознакомиться с теоретическими сведениями.
2. Изучить механизмы управления процессами и потоками.
3. Выполнить следующие задания из файла «zadanie Process» (последней версии). В имена используемых в командах создаваемых файлов должна быть включена фамилия студента (латиницей) во избежание копирования отчётов.

Отчет должен включать:

- Название работы и ее цель;
- Действия, выполняемые по данному руководству и результаты их работы.
- Задания из файла «zadanie Process», команды для их выполнения и результаты их работы.

Приложение 1.

В таблице 2 приведены некоторые сигналы, требуемые стандартом POSIX.

Таблица 2 – Описание сигналов POSIX.

Сигнал	Значение
SIGABRT	Прерывание процесса и создание дампа памяти
SIGALRM	Истекло время будильника
SIGBUS	Ошибка шины при обращении к физической памяти
SIGFPE	Ошибка выполнения операции с плавающей точкой
SIGHUP	Потеря связи процесса с управляющим терминалом пользователя
SIGILL	Попытка выполнить неправильную, несуществующую или привилегированную инструкцию (ill = illegal instruction)
SIGINT	Остановка процесса пользователем с терминала
SIGKILL	Уничтожение процесса, сигнал не может быть проигнорирован или перехвачен
SIGLOST	Потеря блокировки файла
SIGPIPE	Процесс пишет в канал, никем не читаемый (обрыв соединения с читающей стороны)
SIGQUIT	Пользователь выполнил команду «quit» в терминале, создать дамп памяти
SIGSEGV	Процесс обратился к несуществующему адресу памяти или нарушение прав доступа
SIGSTOP	Остановить процесс. Процесс можно возобновить с помощью сигнала SIGCONT
SIGTERM	Запрос на завершение процесса
SIGUSR1	Определяется приложением
SIGUSR2	Определяется приложением

Работа с потоками POSIX (Pthreads).

Данное приложение описывает некоторые функции библиотеки Pthreads (заголовочный файл <pthread.h>).

Таблица 3 – Ряд вызовов функций стандарта Pthreads.

Вызовы, связанные с потоком	Описание
pthread_create	Создание нового потока
pthread_exit	Завершение работы вызывающего потока
pthread_join	Ожидание выхода из указанного потока
pthread_yield	Освобождение ЦП, позволяющее выполниться другому потоку
pthread_attr_init	Создание и инициализация структуры атрибутов потока
pthread_attr_destroy	Удаление структуры атрибутов потока
pthread_attr_setinheritsched	Позволяет определить, как будут установлены параметры планирования потока: путём наследования от потока создателя или в соответствии с содержимым объекта атрибутов
pthread_attr_setschedpolicy	Используется для установки стратегии планирования (см.условия ниже)
pthread_attr_setschedparam	Используется для установки приоритета (см.условия ниже)
pthread_detach()	Открепление потока (по умолчанию все потоки создаются как прикреплённые)
pthread_setcancelstate	Устанавливает поток в состояние, передаваемое параметром, возвращает предыдущее состояние
pthread_setcanceltype	Устанавливает возможность потока продолжать работу после получения запроса на аннулирование
pthread_testcancel	Проверяет наличие необработанных запросов на аннулирование. Если они есть, активизирует процесс аннулирования в точке своего вызова. В противном случае функция продолжает выполнение потока без

	каких-либо последствий. Вызов этой функции можно разместить в любом месте кода потока, которое считается безопасным для его завершения
--	--

Все потоки Pthread имеют определённые свойства. Стандарт определяет следующие атрибуты потока, которые содержит структура атрибутов:

- область видимости;
- размер стека;
- адрес стека;
- приоритет;
- состояние;

Структуры атрибутов хранятся в объектах атрибутов. Объект атрибутов может быть связан с одним или несколькими потоками.

Функция `pthread_attr_setinheritsched` может принимать следующие значения на вход:

`PTHREAD_INHERIT_SCHED` – атрибуты будут наследоваться от потока-создателя, при этом любые атрибуты планирования, устанавливаемые параметром `attr`, будут игнорироваться;

`PTHREAD_EXPLICIT_SCHED` – атрибуты планирования устанавливаются соответственно атрибутам в объекте атрибутов потока. Если функция получила этот аргумент на вход, то для потока можно использовать функции `pthread_attr_setschedpolicy` и `pthread_attr_setschedparam`.

С помощью функции `pthread_attr_setschedpolicy` можно установить стратегию планирования (параметры определены в заголовочном файле `<sched.h>`):

`SCHED_FIFO` – стратегия FIFO («первым прибыл, первым обслужен»);

`SCHED_RR` – стратегия циклического планирования, каждый поток назначается процессору только в течение определённого кванта времени;

`SCHED_OTHER` – стратегия планирования другого типа (определяется реализацией), принимается по умолчанию.

В стандарте Pthread определяются функции и константы для аннулирования потоков. Константы состояний и типов:

`PTHREAD_CANCEL_ENABLE (PTHREAD_CANCEL_DEFERRED)` - отсроченное аннулирование. Эти состояние и тип аннулирования потока устанавливаются по умолчанию. Аннулирование потока происходит, когда он достигает соответствующей точки в своем выполнении или когда программист определяет точку аннулирования с помощью функции `pthread_testcancel()`;

`PTHREAD_CANCEL_ENABLE (PTHREAD_CANCEL_ASYNC)` - асинхронное аннулирование. Аннулирование потока происходит немедленно;

`PTHREAD_CANCEL_DISABLE (любое)` - аннулирование запрещено. Оно вообще не выполняется.

К тому же, стандарт POSIX определяет ряд реентерабельных функций. Блок кода считается реентерабельным, если его невозможно изменить при выполнении. Реентерабельный код исключает возникновение условий «гонок» благодаря отсутствию ссылок на глобальные переменные и модифицируемые статические данные. Следовательно, такой код могут совместно использовать несколько параллельных потоков или процессов без риска создания условий «гонок».

Управление потоками Linux.

В 2000 году в Linux был введен системный вызов `clone`, который размыл отличия между процессами и потоками. Вызова `clone` нет ни в одной другой версии UNIX.

Классически, при создании нового потока исходный поток (потоки) и новый поток совместно использовали все, кроме регистров, — в частности, дескрипторы для открытых файлов, обработчики сигналов, прочие глобальные свойства. Системный вызов `clone` дал возможность все эти аспекты сделать специфичными как для процесса, так и для потока. Формат вызова выглядит следующим образом:

```
pid = clone(function, stack_ptr, sharing_flags, arg);
```

Вызов `clone` создает новый поток либо в текущем процессе, либо в новом процессе (в зависимости от флага `sharing_flags`). Если новый поток находится в текущем процессе, то он совместно с существующими потоками использует адресное пространство процесса и каждая последующая запись в любой байт адресного пространства (любым потоком) тут же становится видима всем остальным потокам процесса.

С другой стороны, если адресное пространство совместно не используется, то новый поток получает точную копию адресного пространства, но последующие записи из нового потока уже не видны старым потокам. Здесь используется та же семантика, что и у системного вызова `fork` по стандарту POSIX.

В обоих случаях новый поток начинает выполнение функции `function` с аргументом `arg` в качестве единственного параметра. Также в обоих случаях новый поток получает свой собственный приватный стек, при этом указатель стека инициализируется параметром `stack_ptr`.

Параметр `sharing_flags` представляет собой битовый массив, обеспечивающий существенно более тонкую настройку совместного

использования, чем традиционные системы UNIX. Каждый бит может быть установлен независимо от остальных и каждый из них определяет, копирует ли новый поток эту структуру данных или использует ее совместно с вызывающим потоком. В таблице 4 показаны значения битов массива `sharing flags`.

Таблица 4 – Биты массива `sharing_flags`.

Флаг	Значение при установке в 1	Значение при установке в 0
<code>CLONE_VM</code>	Создать новый поток	Создать новый процесс
<code>CLONE_FS</code>	Общие рабочий каталог, каталог <code>root</code> и <code>umask</code>	Не использовать их совместно
<code>CLONE_FILES</code>	Общие дескрипторы файлов	Копировать дескрипторы файлов
<code>CLONE.SIGHAND</code>	Общая таблица обработчика сигналов	Копировать таблицу
<code>CLONE_PID</code>	Новый поток получает старый PID	Новый поток получает свой собственный PID
<code>CLONE_PARENT</code>	Новый поток имеет того же родителя, что и вызывающий	Родителем нового потока является вызывающий

Такая детализация вопросов совместного использования стала возможна благодаря тому, что в системе Linux для различных элементов, (параметры планирования, образ памяти и т. д.), используются отдельные структуры данных. Структура задач просто содержит указатели на эти структуры данных, поэтому легко создать новую структуру задач для каждого клонированного потока и сделать так, чтобы она указывала либо на структуры (планирования потоков, памяти и пр.) старого потока, либо на копии этих структур. Сам факт возможности такой высокой степени детализации совместного использования еще не означает, что она полезна — особенно потому, что в традиционных версиях UNIX это не поддерживается. Если какая-либо программа в системе Linux пользуется этой возможностью, то это означает, что она больше не является переносимой на UNIX.

Модель потоков Linux порождает еще одну трудность. UNIX-системы связывают с процессом один PID (независимо от того, однопоточный он или многопоточный). Чтобы сохранять совместимость с другими UNIX-

системами, Linux различает идентификаторы процесса PID и идентификаторы потока TID. Оба этих поля хранятся в структуре задач. Когда вызов clone используется для создания нового процесса (который ничего не использует совместно со своим создателем), PID устанавливается в новое значение; в противном случае задача получает новый TID, но наследует PID. Таким образом, все потоки процесса получают тот же самый PID, что и первый поток процесса.