8 Подробное рассмотрение процессов и использования ресурсов

Эта глава более подробно расскажет вам о взаимоотношениях между процессами, ядром и системными ресурсами. Существует три основных типа аппаратных ресурсов: центральный процессор, память и устройства ввода/вывода. Процессы соперничают за эти ресурсы, и задачей ядра является их честное распределение. Само ядро также является ресурсом — программным ресурсом, который процессы используют, чтобы выполнять такие задачи, как создание новых процессов и взаимодействие с другими процессами.

Многие из инструментов, которые вы увидите в этой главе, часто считают инструментами для слежения за производительностью. Они чрезвычайно полезны, если ваша система начинает «тормозить» и вы пытаетесь выяснить причину этого. Тем не менее не следует уделять излишнего внимания производительности: попытки оптимизировать систему, которая и так работает хорошо, зачастую являются лишь пустой тратой времени. Вместо этого сосредоточьтесь на понимании того, что́ эти инструменты измеряют на самом деле, и тогда вы получите прекрасное представление о том, как работает ядро.

8.1. Отслеживание процессов

Из раздела 2.16 вы узнали о том, как использовать команду `ps`, чтобы увидеть перечень процессов, запущенных в системе в данный момент. Команда `ps` приводит список текущих процессов, но она может мало что сказать вам о том, как изменяются процессы с течением времени. Следовательно, она не поможет вам определить процесс, который использует слишком много ресурсов ЦПУ или оперативной памяти.

Команда `top` часто оказывается полезнее команды `ps`, поскольку она отображает текущее состояние системы, заодно со многими полями, которые есть в листинге команды `ps`, и при этом она обновляет результат каждую секунду. Вероятно, самым важным является то, что команда `top` показывает наиболее активные процессы (то есть те, которые в данный момент используют наибольшую часть процессорного времени) в верхней части списка.

Нажимая на клавиши, можно отправлять инструкции команде top. Приведу самые важные из них (табл. 8.1).

Таблица 8.1. Инструкции для команды top

Инструкция	Действие
Клавиша Пробел	Немедленно обновить экран
M	Выполнить сортировку по количеству используемой резидентной памяти
T	Выполнить сортировку по общему (кумулятивному) применению ЦПУ
P	Выполнить сортировку по текущему использованию ЦПУ (по умолчанию)
u	Отобразить процессы только для одного пользователя
f	Выбрать другие параметры для отображения
?	Отобразить статистику использования всех команд top

Две другие утилиты Linux, подобные команде top, — atop и htop — предлагают расширенный набор вариантов просмотра и функций. Большинство дополнительных функций доступно в других утилитах. Например, команда htop обладает многими возможностями команды lsof, описанной в следующем разделе.

8.2. Поиск открытых файлов с помощью команды lsof

Команда lsof перечисляет открытые файлы и процессы, которые их используют. Поскольку Unix делает существенный акцент на файлах, команда lsof входит в число самых полезных инструментов для отыскания неполадок. Однако эта команда не ограничивается обычными файлами — она может перечислять сетевые ресурсы, динамические библиотеки, каналы и многое другое.

8.2.1. Чтение результатов вывода команды lsof

После запуска команды lsof в командной строке обычно появляется огромный список. Ниже приведен фрагмент того, что вы могли бы увидеть. Этот результат содержит файлы, открытые процессом init, а также запущенный процесс vi:

```
$ lsof
COMMAND PID  USER  FD TYPE DEVICE  SIZE     NODE NAME
init      1  root cwd  DIR    8,1  4096        2 /
init      1  root rtd  DIR    8,1  4096        2 /
init      1  root mem  REG    8,  47040  9705817 /lib/i386-linux-gnu/libnss_files-
                                                  2.15.so
init      1  root mem  REG    8,1 42652  9705821 /lib/i386-linux-gnu/libnss_nis-
                                                  2.15.so
init      1  root mem  REG    8,1 92016  9705833 /lib/i386-linux-gnu/libnsl-2.15.so
--snip--
vi    22728 juser cwd  DIR    8,1  4096 14945078 /home/juser/w/c
vi    22728 juser  4u  REG    8,1  1288  1056519 /home/juser/w/c/f
--snip--
```

Результат состоит из следующих полей (перечисленных в верхней строке).

- COMMAND. Командное имя для процесса, который удерживает дескриптор файла.
- PID. Идентификатор процесса.
- USER. Пользователь, запустивший процесс.
- FD. Это поле может содержать два типа элементов. В приведенном выше результате столбец FD показывает назначение файла. Это поле может также содержать *файловый дескриптор* открытого файла — число, которое процесс использует вместе с системными библиотеками и ядром, чтобы идентифицировать файл и работать с ним.
- TYPE. Тип файла (обычный файл, каталог, сокет и т. п.).
- DEVICE. Старший и младший номера устройства, которое удерживает данный файл.
- SIZE. Размер файла.
- NODE. Номер дескриптора inode для данного файла.
- NAME. Имя файла.

Страница руководства lsof(1) содержит полный перечень того, что вы можете встретить в каждом из полей, однако вы должны уметь определять, что перед вами, просто глядя на результат вывода. Посмотрите, например, на записи, у которых в поле FD указано значение cwd (выделено жирным шрифтом). В этих строках заданы текущие рабочие каталоги для процессов. Еще один пример содержится в самой последней строке: это файл, который в данный моммент редактируется пользователем с помощью команды vi.

8.2.2. Использование команды lsof

Есть два основных подхода к запуску команды lsof.

- Перечислить все, а затем перенаправить вывод в команду типа less и поискать то, что вам необходимо. На это может потребоваться некоторое время, в зависимости от результата вывода.
- Сузить список, создаваемый командой lsof, с помощью параметров командной строки.

Можно использовать параметры командной строки, чтобы передать имя файла в качестве аргумента и вынудить команду lsof перечислить только те записи, которые соответствуют этому аргументу. Например, следующая команда отображает записи для файлов, открытых в каталоге /usr:

```
$ lsof /usr
```

Чтобы вывести список файлов, открытых процессом с идентификатором PID, запустите такую команду:

```
$ lsof -p pid
```

Для вывода краткой справки о параметрах команды lsof запустите команду lsof -h. Большинство параметров относится к формату вывода (см. главу 10, в которой говорится о сетевых функциях команды lsof).

ПРИМЕЧАНИЕ

Команда lsof сильно зависит от информации о ядре. Если вы обновляете ядро, но при этом нерегулярно обновляете все остальное, вам может потребоваться обновление команды lsof. Более того, если вы применили обновление и для ядра, и для команды lsof, обновленная команда lsof может не запускаться до тех пор, пока вы не перезагрузите систему с новым ядром.

8.3. Отслеживание выполнения команд и системных вызовов

Инструменты, которые мы рассмотрели, исследуют активные процессы. Однако если вам непонятно, почему какая-либо программа закрывается практически сразу после запуска, то даже команда `lsof` вам не поможет. На самом деле у вас возникли бы сложности, если бы вы запустили команду `lsof` одновременно с командой, вызывающей отказ.

Команды `strace` (отслеживание системных вызовов) и `ltrace` (отслеживание библиотек) могут помочь выяснить, что пытается делать команда. Эти инструменты выводят чрезвычайно большие отчеты, но как только вы узнаете, что искать, в вашем распоряжении будут дополнительные инструменты для отслеживания проблем.

8.3.1. Команда strace

Вспомните о том, что *системный вызов* является привилегированной операцией, которую процесс из пространства пользователя просит у ядра выполнить (например, открытие файла и чтение данных из него). Утилита `strace` выводит список всех системных вызовов, которые осуществляет процесс. Чтобы увидеть это в действии, запустите такую команду:

```
$ strace cat /dev/null
```

Из главы 1 вы узнали о том, что, когда процесс собирается запустить другой процесс, он задействует системный вызов `fork()`, чтобы создать ответвленную копию, которая затем использует один из системных вызовов семейства `exec()`, чтобы запустить новую команду. Команда `strace` начинает работать с новым процессом (копией исходного процесса) сразу после вызова `fork()`. Следовательно, первые строки вывода данной команды должны показать команду `execve()` в действии, за которой следует вызов инициализации памяти, `brk()`, как приведено ниже:

```
execve("/bin/cat", ["cat", "/dev/null"], [/* 58 vars */]) = 0
brk(0)                                  = 0x9b65000
```

Следующая часть вывода относится главным образом к загрузке совместно используемых библиотек. Можете пропустить это, если вы не стремитесь узнать о том, что делает система совместно используемых библиотек.

```
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0xb77b5000
```

```
access("/etc/ld.so.preload", R_OK)    = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
--snip--
open("/lib/libc.so.6", O_RDONLY)       = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\200^\1"..., 1024)= 1024
```

Кроме того, пропустите вывод до команды `mmap` включительно, пока не встретите строки, подобные следующим:

```
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 6), ...}) = 0
open("/dev/null", O_RDONLY|O_LARGEFILE) = 3
fstat64(3, {st_mode=S_IFCHR|0666, st_rdev=makedev(1, 3), ...}) = 0
fadvise64_64(3, 0, 0, POSIX_FADV_SEQUENTIAL)= 0
read(3,"", 32768)                      = 0
close(3)                               = 0
close(1)                               = 0
close(2)                               = 0
exit_group(0)                          = ?
```

Эта часть вывода показывает команду в действии. Сначала посмотрите на вызов `open()`, который открывает файл. Число 3 — результат, означающий успешное завершение (это файловый дескриптор, который ядро возвращает после открытия файла). Под ним вы видите, где команда `cat` выполняет чтение из устройства `/dev/null` (вызов `read()`, который также обладает файловым дескриптором `3`). Считывать больше нечего, поэтому команда закрывает файловый дескриптор и выходит с помощью вызова `exit_group()`.

Что происходит, если возникает ошибка? Попробуйте запустить команду `strace cat not_a_file` и посмотрите на системный вызов `open()` в результатах вывода:

```
open("not_a_file", O_RDONLY|O_LARGEFILE) = -1 ENOENT (No such file or directory)
```

Поскольку команде `open()` не удалось открыть файл, она возвратила значение `-1`, чтобы сообщить об ошибке. Видно, что команда `strace` выводит название ошибки и дает ее краткое описание.

Отсутствующие файлы являются наиболее частым источником ошибок в командах Unix, поэтому если системный журнал и другая информация оказываются не слишком полезными, а обратиться больше не к чему, то команда `strace` может оказать существенную помощь. Ее можно применить даже для демонов, которые откреплены. Например, так:

```
$ strace -o crummyd_strace -ff crummyd
```

В данном примере параметр `-o` команды `strace` заносит в журнал действия любого дочернего процесса, который демон `crummy` породил в `crummyd_strace.`*`pid`*, где *`pid`* — это идентификатор дочернего процесса.

8.3.2. Команда ltrace

Команда `ltrace` отслеживает вызовы совместно используемых библиотек. Результаты ее работы напоминают вывод команды `strace`, и именно поэтому я упоминаю

о ней здесь, но она не отслеживает ничего на уровне ядра. Имейте в виду: вызовов совместно используемых библиотек *намного* больше, чем системных вызовов. Вам непременно понадобится фильтровать результаты, и у команды `ltrace` есть множество встроенных параметров, чтобы помочь вам в этом.

ПРИМЕЧАНИЕ

См. подраздел 15.1.4, содержащий дополнительную информацию. Команда ltrace не работает для статически связанных двоичных файлов.

8.4. Потоки

В Linux некоторые процессы разделены на части, называемые *потоками*. Поток очень похож на процесс: у него есть идентификатор (TID, или ID потока), и ядро планирует запуск потоков и запускает их так же, как и процессы. Однако в отличие от отдельных процессов, которые обычно не используют совместно с другими процессами такие системные ресурсы, как оперативная память и подключение к вводу/выводу, все потоки внутри какого-либо процесса совместно задействуют ресурсы системы и некоторую часть памяти.

8.4.1. Однопоточные и многопоточные процессы

Многие процессы обладают только одним потоком. Процесс с одним потоком является *однопоточным*, а процесс с несколькими потоками — *многопоточным*. Все процессы запускаются как однопоточные. Этот стартовый поток обычно называется *главным потоком*. Затем главный поток может запустить новые потоки, чтобы процесс стал многопоточным, подобно тому как процесс может вызвать команду `fork()` для запуска нового процесса.

ПРИМЕЧАНИЕ

Если процесс является однопоточным, то о потоках вообще довольно редко упоминают. В этой книге на потоки не обращается внимание, если многопоточные процессы не отражаются на том, что вы видите или осуществляете.

Основное преимущество многопоточных процессов таково: когда процесс должен выполнить много работы, потоки могут быть запущены одновременно на нескольких процессорах, что потенциально ускоряет вычисления. Хотя одновременные вычисления можно организовать и с помощью нескольких процессов, потоки запускаются быстрее процессов и потокам часто бывает проще и/или эффективнее взаимодействовать между собой при совместном использовании памяти по сравнению с процессами, которые взаимодействуют через сетевое соединение или канал.

Некоторые команды применяют потоки, чтобы обойти проблемы при управлении несколькими ресурсами ввода/вывода. Традиционно процесс использовал бы что-либо вроде команды `fork()`, чтобы запустить новый подпроцесс для работы с новым потоком ввода или вывода. Потоки предлагают похожий механизм без излишнего запуска нового процесса.

8.4.2. Просмотр потоков

По умолчанию в выводе команд ps и top отображаются только процессы. Чтобы показать информацию о потоке в команде ps, добавьте параметр m (пример 8.1).

Пример 8.1. Просмотр потоков с помощью команды ps m

```
$ ps m
  PID TTY      STAT   TIME COMMAND
 3587 pts/3    -      0:00 bash❶
    - -        Ss     0:00 -
 3592 pts/4    -      0:00 bash❷
    - -        Ss     0:00 -
12287 pts/8    -      0:54 /usr/bin/python /usr/bin/gm-notify❸
    - -        SL1    0:48 -
    - -        SL1    0:00 -
    - -        SL1    0:06 -
    - -        SL1    0:00 -
```

В примере 8.1 процессы показаны вместе с потоками. Каждая строка с номером в столбце PID (эти строки отмечены символами ❶, ❷ и ❸) представляет процесс как при обычном выводе команды ps. Строки с дефисами в столбце PID представляют потоки, связанные с данным процессом. В этом выводе у каждого из процессов ❶ и ❷ только один поток, а процесс 12287 (❸) является многопоточным и состоит из четырех потоков.

Если вы желаете просмотреть идентификаторы потоков с помощью команды ps, можно использовать специальный формат вывода. В примере 8.2 показаны только идентификаторы процессов и потоков, а также сама команда.

Пример 8.2. Отображение идентификаторов процессов и потоков с помощью команды ps m

```
$ ps m -o pid,tid,command
  PID   TID   COMMAND
 3587     -   bash
    -  3587   -
 3592     -   bash
    -  3592   -
 12287    -   /usr/bin/python /usr/bin/gm-notify
    - 12287   -
    - 12288   -
    - 12289   -
    - 12295   -
```

Приведенный в примере 8.2 вывод соответствует потокам, показанным в примере 8.1. Обратите внимание на то, что идентификаторы потоков для однопоточных процессов совпадают с идентификаторами процессов: это главные потоки. Для многопоточного процесса 12287 поток 12287 также является главным потоком.

ПРИМЕЧАНИЕ

Как правило, вам не придется взаимодействовать с отдельными потоками так, как вы это делали бы с процессами. Вам потребуется узнать довольно много о том, как была написана многопоточная команда, чтобы воздействовать на один из потоков в какой-либо момент, но даже в этом случае такая идея не слишком хороша.

Потоки могут вызвать путаницу при отслеживании ресурсов, поскольку отдельные потоки в многопоточном процессе могут одновременно пользоваться ресурсами. Например, команда `top` по умолчанию не отображает потоки; необходимо нажать клавишу H, чтобы включить их показ. Для большинства инструментов отслеживания ресурсов, о которых вы скоро узнаете, потребуется выполнить небольшую дополнительную работу, чтобы включить отображение потоков.

8.5. Введение в отслеживание ресурсов

Сейчас мы обсудим некоторые моменты, относящиеся к отслеживанию ресурсов, включая время центрального процессора, память и дисковый ввод/вывод. Мы рассмотрим использование ресурсов как в масштабе всей системы, так и для отдельных процессов.

Многие пользователи вникают в устройство ядра системы Linux в целях улучшения производительности. Однако большинство версий систем прекрасно работает с установками по умолчанию, и вы можете потратить несколько дней, пытаясь настроить производительность компьютера без существенных результатов, особенно если вы не знаете, что искать. По этой причине, когда вы будете экспериментировать с инструментами, описанными в этой главе, думайте не о производительности, а о том, как действует ядро, распределяя ресурсы между процессами.

8.6. Измерение процессорного времени

Чтобы отследить один или несколько процессов с течением времени, используйте параметр `-p` в команде `top` с таким синтаксисом:

```
$ top -p pid1 [-p pid2 ...]
```

Чтобы выяснить, какое количество процессорного времени применяет команда для своей работы, используйте команду `time`. В большинстве оболочек есть встроенная команда `time`, которая не приводит подробную статистику, поэтому может потребоваться запуск команды `/usr/bin/time`. Например, чтобы измерить процессорное время, использованное командой `ls`, запустите такую команду:

```
$ /usr/bin/time ls
```

По окончании работы команды `ls` команда `time` должна вывести результаты, подобные приведенным ниже. Ключевые поля выделены жирным шрифтом:

```
0.05user 0.09system 0:00.44elapsed 31%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (125major+51minor)pagefaults 0swaps
```

- **Время пользователя.** Количество секунд, которое центральный процессор потратил на выполнение собственного кода команды. В современных процессорах некоторые команды запускаются настолько быстро и процессорное время настолько мало, что значение округляется до нуля.
- **Время системы.** Какое количество времени ядро затрачивает на выполнение работы процесса (например, на чтение файлов и каталогов).

- **Время работы.** Общее количество времени, которое требуется на работу процесса, от его запуска до завершения, включая время, затраченное процессором на выполнение других задач. Эта величина, как правило, не слишком пригодна для измерения производительности, однако, если вычесть из нее значения времени пользователя и времени системы, можно получить общее представление о том, как долго процесс пребывает в ожидании системных ресурсов.

Остальная часть вывода содержит главным образом подробности об использовании памяти и ресурсов ввода/вывода. Подробнее об ошибках отсутствия страницы вы прочитаете в разделе 8.9.

8.7. Настройка приоритетов процессов

Можно изменить распорядок, который ядро назначает процессам, чтобы предоставить какому-либо процессу больше или меньше процессорного времени по сравнению с другими процессами. Ядро запускает каждый процесс в соответствии с назначенным ему *приоритетом*, который является числом от –20 до 20, причем –20 означает высший приоритет. (Да, это сбивает с толку!)

Команда `ps -l` выводит текущий приоритет процесса, однако немного проще увидеть приоритеты в действии с помощью команды `top`, как показано здесь:

```
$ top
Tasks: 244 total,  2 running, 242 sleeping,   0 stopped,   0 zombie
Cpu(s): 31.7%us, 2.8%sy,  0.0%ni, 65.4%id,  0.2%wa,  0.0%hi,  0.0%si,   0.0%st
Mem:   6137216k total, 5583560k used,   553656k free,    72008k buffers
Swap:  4135932k total,  694192k used, 3441740k  free,   767640k cached
  PID  USER     PR NI  VIRT  RES  SHR S %CPU %MEM    TIME+ COMMAND
28883  bri      20  0 1280m 763m  32m S   58 12.7 213:00.65 chromium-
browse
 1175  root     20  0  210m  43m  28m R   44  0.7  14292:35 Xorg
 4022  bri      20  0  413m 201m  28m S   29  3.4   3640:13 chromium-browse
 4029  bri      20  0  378m 206m  19m S    2  3.5  32:50.86 chromium-browse
 3971  bri      20  0  881m 359m  32m S    2  6.0 563:06.88 chromium-browse
 5378  bri      20  0  152m  10m 7064 S    1  0.2  24:30.21 compiz
 3821  bri      20  0  312m  37m  14m S    0  0.6  29:25.57 soffice.bin
 4117  bri      20  0  321m 105m  18m S    0  1.8  34:55.01 chromium-browse
 4138  bri      20  0  331m  99m  21m S    0  1.7 121:44.19 chromium-browse
 4274  bri      20  0  232m  60m  13m S    0  1.0  37:33.78 chromium-browse
 4267  bri      20  0 1102m 844m 11m S    0 14.1  29:59.27 chromium-browse
 2327  bri      20  0  301m   43m 16m S    0  0.7 109:55.65 unity-2d-shell
```

В приведенном отчете команды `top` столбец `PR` («приоритет») содержит текущий приоритет ядра для запуска данного процесса. Чем больше число, тем меньше вероятность того, что ядро запланирует этот процесс, если процессорное время необходимо другим процессам. Однако один лишь приоритет запуска не определяет решение ядра о предоставлении процессорного времени процессу, к тому же он часто меняется во время выполнения команды в соответствии с количеством процессорного времени, потребляемого процессом.

За столбцом приоритета следует столбец, содержащий *значение относительного приоритета* (NI), которое дает рекомендацию планировщику в ядре. Об этом значении следует позаботиться, если вы пытаетесь повлиять на решение ядра. Ядро прибавляет значение относительного приоритета к текущему приоритету, чтобы определить следующий квант времени для данного процесса.

По умолчанию значение относительного приоритета равно 0. Допустим, что вы запускаете в фоновом режиме большой объем вычислений и желаете, чтобы он не замедлял вашу работу в интерактивном сеансе. Чтобы такой процесс занял последнее место по отношению к другим процессам и работал лишь тогда, когда остальным задачам нечего делать, можно установить для него значение относительного приоритета равным 20 с помощью команды `renice` (здесь параметр *pid* — идентификатор процесса, который вы желаете изменить):

```
$ renice 20 pid
```

Если вы пользователь-superuser, можно указать отрицательное значение относительного приоритета, но такая идея практически всегда является вредной, поскольку для системных процессов может не оказаться достаточного количества процессорного времени. На самом деле вам, вероятно, не потребуется часто менять значения относительного приоритета, поскольку во многих системах Linux всего один пользователь, который не выполняет большие объемы вычислений. Значение относительного приоритета было гораздо более важным тогда, когда на одном компьютере работало несколько пользователей.

8.8. Средние значения загрузки

Производительность процессора — один из самых простых параметров для измерения. *Среднее значение загрузки* является средним количеством процессов, которые в данный момент готовы к запуску. То есть это оценка числа процессов, способных использовать процессор в конкретный момент времени. Осмысляя это значение, имейте в виду, что большинство процессов в системе обычно ожидает ввода (с клавиатуры, с помощью мыши или из сети, например), и это означает, что большинство процессов не готовы к запуску и ничего не вносят в среднее значение загрузки. Только те процессы, которые действительно что-либо выполняют, влияют на среднее значение загрузки.

8.8.1. Использование команды uptime

Команда `uptime` сообщает три средних значения загрузки в дополнение к тому, как долго работает ядро:

```
$ uptime
... up 91 days, ... load average: 0.08, 0.03, 0.01
```

Три числа, выделенных жирным шрифтом, являются средними значениями загрузки за последние 1, 5 и 15 минут соответственно. Как видите, система не очень занята: за последние 15 минут на всех процессорах работало в среднем только

0,01 процесса. Другими словами, если бы у вас был всего один процессор, то он запускал бы приложения из пространства пользователя лишь 1 % времени за последние 15 минут. Традиционно в большинстве ПК среднее значение загрузки — около 0, если вы заняты чем-либо отличным от компилирования программы или компьютерной игры. Нулевое значение обычно является хорошим признаком, поскольку оно означает, что ваш процессор не перегружен и вы экономите мощность.

ПРИМЕЧАНИЕ

В современных ПК компоненты пользовательского интерфейса стремятся использовать больше ресурсов процессора, чем это было раньше. Например, в системах Linux плагин Flash для браузера приобрел печальную известность как пожиратель ресурсов, а Flash-приложения могут с легкостью занять большую часть процессорного времени и памяти вследствие некачественной реализации в целом.

Если среднее значение загрузки достигает 1, то, вероятно, один процесс практически постоянно использует центральный процессор. Чтобы выяснить, что это за процесс, воспользуйтесь командой `top`; этот процесс будет показан на экране в самой верхней части списка.

В большинстве современных систем присутствует несколько процессорных ядер или процессоров, поэтому несколько процессов могут легко работать одновременно. Если ядер два, то среднее значение загрузки 1 означает, что только одно из ядер активно в любой момент времени, а среднее значение 2 говорит о том, что оба ядра работают.

8.8.2. Высокие значения загрузки

Высокое среднее значение загрузки не обязательно свидетельствует о том, что система испытывает сложности. Система, у которой достаточное количество оперативной памяти и ресурсов ввода/вывода, способна легко справиться с несколькими запущенными процессами. Если среднее значение загрузки высоко, но при этом система нормально откликается, не волнуйтесь: множество процессов совместно используют процессор. Процессы вынуждены соперничать друг с другом за процессорное время, в результате им требуется больше времени на выполнение своих вычислений по сравнению с тем случаем, когда им позволено применять процессор постоянно. Еще один случай, при котором можно высокое среднее значение загрузки является нормальным, это веб-сервер, в котором процессы могут запускаться и прекращаться настолько быстро, что функция измерения средней загрузки не может эффективно работать.

Тем не менее, если вы чувствуете, что система «тормозит» и значение средней загрузки велико, вероятно, присутствуют проблемы с производительностью оперативной памяти. Когда в системе мало памяти, ядро может начать быстро подкачивать с диска память для процессов. Когда такое происходит, многие процессы становятся готовы к запуску, но при этом для них может быть недоступна память, и тогда они остаются в состоянии готовности (и вносят вклад в среднее значение загрузки) намного дольше, чем при нормальном режиме работы.

Сейчас мы более подробно рассмотрим оперативную память.

8.9. Память

Один из самых простых способов проверить состояние системной памяти в целом — запустить команду `free` или посмотреть файл `/proc/meminfo`, чтобы понять, сколько реальной памяти используется для кэша и буферов. Как мы только что отмечали, проблемы с производительностью могут быть вызваны недостатком памяти. Если используется немного памяти для кэша/буфера (и вместо этого расходуется реальная память), вам может понадобиться дополнительная память. Однако было бы чересчур просто полагать, что проблемы с производительностью вызваны лишь недостатком памяти.

8.9.1. Как работает память

В процессоре присутствует модуль управления памятью (MMU), который переводит виртуальные адреса памяти, используемые процессами, в реальные. Ядро помогает модулю MMU, разбивая память на маленькие фрагменты, называемые *страницами*. Ядро содержит структуру данных, которая называется *таблицей страниц* и содержит схему соответствия виртуальных адресов страниц реальным адресам страниц в памяти. Когда процесс получает доступ к памяти, модуль MMU переводит виртуальные адреса, используемые процессом, в реальные адреса на основе таблицы страниц ядра.

В действительности пользовательскому процессу для работы не нужны сразу все его страницы. Обычно ядро загружает и распределяет страницы по мере их необходимости для процесса; такая система работы известна как *вызов страниц по запросу* или *листание по запросу*. Чтобы понять, как это устроено, рассмотрим запуск и работу команды в качестве нового процесса.

1. Ядро загружает начало кода с инструкциями команды в страницы памяти.
2. Ядро может выделить несколько страниц рабочей памяти для нового процесса.
3. Во время своей работы процесс может дойти до такого момента, когда следующей инструкции не окажется ни в одной из страниц, загруженных ядром изначально. Тогда ядро вступает в действие, загружает необходимые страницы в память и позволяет команде продолжить выполнение.
4. Подобным же образом, если команде необходимо больше рабочей памяти, чем было выделено изначально, ядро решает эту задачу, отыскивая свободные страницы (или освобождая пространство) и назначая их данному процессу.

8.9.2. Ошибки из-за отсутствия страниц

Если страница памяти не готова, когда процессу необходимо ее использовать, процесс вызывает *ошибку из-за отсутствия страницы*. При возникновении такого события ядро забирает у процесса управление процессором, чтобы подготовить страницу. Существуют два типа ошибок из-за отсутствия страниц: малые и большие.

Малые ошибки

Малая ошибка из-за отсутствия страницы возникает тогда, когда желаемая страница фактически находится в основной памяти, но модуль MMU не знает, где она. Такое может произойти, когда процесс требует больше памяти или когда модуль MMU не обладает достаточным пространством для хранения всех местоположений страниц для процесса. В таком случае ядро сообщает модулю MMU данные о странице и позволяет процессу продолжить работу. Малые ошибки из-за отсутствия страницы не столь уж существенны, и многие из них возникают во время работы процесса. Если вам не требуется максимальная производительность какой-либо программы, интенсивно обращающейся к памяти, вероятно, не стоит беспокоиться об этих ошибках.

Большие ошибки

Большая ошибка из-за отсутствия страницы возникает тогда, когда желаемая страница памяти не находится в основной памяти и, значит, ядро должно загрузить ее с диска или из какого-либо другого медленного хранилища данных. Большое количество таких ошибок сильно замедлит работу системы, поскольку ядро должно выполнить довольно солидную работу по снабжению процессов страницами, лишая нормальные процессы возможности работать.

Некоторых больших ошибок невозможно избежать: например, когда код загружается с диска при запуске программы в первый раз. Самые серьезные проблемы возникают, когда памяти становится недостаточно и ядро начинает подкачивать страницы из рабочей памяти на диск, чтобы освободить место для новых страниц.

Отслеживание ошибок из-за отсутствия страниц

Можно отследить ошибки страниц для отдельных процессов с помощью команд `ps`, `top` и `time`. Следующая команда показывает простой пример того, как команда `time` отображает ошибки страниц. Результаты работы команды `cal` не имеют значения, поэтому мы их отключили, перенаправив в устройство `/dev/null`.

```
$ /usr/bin/time cal > /dev/null
0.00user 0.00system 0:00.06elapsed 0%CPU (0avgtext+0avgdata 3328maxresident)k
648inputs+0outputs (2major+254minor)pagefaults 0swaps
```

Как можно заметить из выделенного жирным шрифтом текста, при работе программы произошли две большие и 254 малые ошибки из-за отсутствия страниц. Большие ошибки возникли, когда ядру потребовалось загрузить команду с диска в первый раз. Если бы вы запустили эту команду повторно, то больших ошибок, вероятно, не было бы, поскольку ядро выполнило кэширование страниц с диска.

Если вы хотите увидеть ошибки для работающих процессов, воспользуйтесь командой `top` или `ps`. При запуске команды `top` используйте флаг `f`, чтобы изменить отображаемые поля, и флаг `u`, чтобы отобразить количество больших ошибок. Результаты будут показаны в новом столбце, `nFLT`. Количество малых ошибок вы не увидите.

При использовании команды `ps` можно применить специальный формат вывода, чтобы увидеть ошибки для конкретного процесса. Вот пример для процесса с идентификатором ID 20365:

```
$ ps -o pid,min_flt,maj_flt 20365
  PID  MINFL  MAJFL
20365 834182     23
```

Столбцы MINFL и MAJFL показывают число малых и больших ошибок из-за отсутствия страниц. Конечно, можно сочетать все это с любыми другими параметрами выбора процессов, как рассказано на странице руководства ps(1).

Просмотр страниц ошибок по процессам может помочь вам сконцентрироваться на отдельных проблематичных компонентах. Тем не менее, если вы заинтересованы в производительности системы в целом, вам необходим инструмент, позволяющий вывести итог по использованию процессора и памяти всеми процессами.

8.10. Отслеживание производительности процессора и памяти с помощью команды vmstat

Среди множества инструментов, доступных для отслеживания производительности, команда vmstat является одним из самых старых, содержащих минимум необходимой информации. Она пригодится для получения высокоуровневого представления о том, как часто ядро выполняет подкачку страниц, насколько загружен процессор и как используются ресурсы ввода/вывода.

Хитрость в овладении мощью команды vmstat состоит в понимании ее отчета. Вот, например, результаты работы команды vmstat 2, которая сообщает статистику каждые две секунды:

```
$ vmstat 2
procs -----------memory---------- ---swap-- -----io---- -system-- ----cpu----
 r  b   swpd   free   buff    cache  si  so   bi   bo  in  cs us sy id wa
 2  0 320416 3027696 198636 1072568   0   0    1    1   2   0 15  2 83  0
 2  0 320416 3027288 198636 1072564   0   0    0 1182 407 636  1  0 99  0
 1  0 320416 3026792 198640 1072572   0   0    0   58 281 537  1  0 99  0
 0  0 320416 3024932 198648 1074924   0   0    0  308 318 541  0  0 99  1
 0  0 320416 3024932 198648 1074968   0   0    0    0 208 416  0  0 99  0
 0  0 320416 3026800 198648 1072616   0   0    0    0 207 389  0  0 100 0
```

Этот вывод распределяется по таким категориям: procs — для процессов, memory — для использования памяти, swap — для страниц, которые перемещаются в область подкачки и из нее, io — для использования диска, system — для количества переключений ядра на его код и cpu — для количества времени, затраченного различными частями системы.

Приведенный выше пример типичен для систем, которые не выполняют много работы. Обычно следует начинать просмотр со второй строки — в первой содержатся средние значения за все время работы системы. Например, в данном случае система переместила на диск (swpd) 320 416 Кбайт памяти, при этом свободно около 3 025 000 Кбайт (3 Гбайт) реальной памяти. Хотя некоторая часть области подкачки использована, нулевые значения в столбцах si (swap-in, «входящая» подкачка) и so (swap-out, «выходящая» подкачка) говорят о том, что в данный момент

ядро не занято никаким из видов подкачки с диска. Столбец `buff` сообщает объем памяти, который ядро использует для дисковых буферов (см. подраздел 4.2.5).

В правом столбце с заголовком `CPU` можно увидеть распределение процессорного времени (столбцы `us`, `sy`, `id` и `wa`). Они сообщают соответственно процентное соотношение времени, которое процессор тратит на задачи пользователя, системные задачи (задачи ядра), бездействие и ожидание ввода/вывода. В приведенном примере запущено не так много пользовательских процессов (они используют не более 1 % процессорного времени); ядро не делает практически ничего, в то время как процессор находится в бездействии 99 % всего времени.

Теперь взгляните, что происходит, если через некоторое время запускается большая команда (первые две строки появились перед самым запуском программы) (пример 8.3).

Пример 8.3. Активность памяти

```
procs -----------memory---------- ---swap-- -----io---- -system-- ----cpu----
r b     swpd   free   buff     cache  si  so    bi   bo   in   cs us sy id wa
1 0   320412  2861252  198920 1106804   0   0     0    0 2477 4481 25 2 72 0❶
1 0   320412  2861748  198924 1105624   0   0     0   40 2206 3966 26 2 72 0
1 0   320412  2860508  199320 1106504   0   0   210   18 2201 3904 26 2 71 1
1 1   320412  2817860  199332 1146052   0   0 19912    0 2446 4223 26 3 63 8
2 2   320284  2791608  200612 1157752 202   0  4960  854 3371 5714 27 3 51 18❷
1 1   320252  2772076  201076 1166656  10   0  2142 1190 4188 7537 30 3 53 14
0 3   320244  2727632  202104 1175420  20   0  1890  216 4631 8706 36 4 46 14
```

Как следует из примера 8.3 (маркер ❶), процессор используется в течение продолжительного периода, в особенности пользовательскими процессами. Поскольку свободной памяти достаточно, объем использованного кэша и буфера начинает возрастать, так как ядро применяет диск сильнее.

Чуть позже можно увидеть интересное (маркер ❷): ядро извлекает в память страницы из области подкачки (столбец `si`). Это означает, что команда, которая только что запустилась, запросила некоторые из страниц, используемых совместно с другим процессом. Такое встречается часто, многие процессы применяют код из определенных общих библиотек только при своем запуске.

Обратите также внимание на то, что столбец `b` сообщает о том, что некоторые процессы *блокированы* (им не разрешен запуск) в ожидании страниц памяти. В целом количество свободной памяти уменьшается, но до ее нехватки еще очень далеко. Наблюдается также значительное количество дисковой активности, что отмечено увеличением значений в столбцах `bi` (blocks in, блоки «на входе») и `bo` (blocks out, блоки «на выходе»).

Результат будет совсем другим, если возникнет нехватка памяти. По мере уменьшения свободного пространства будут уменьшаться и размеры буфера с кэшем, поскольку ядру все в большей степени требуется пространство для пользовательских процессов. Когда не останется совсем ничего, вы увидите активность в столбце `so` («выходящая» подкачка), так как ядро начинает перемещать страницы на диск. В этот момент практически все остальные столбцы вывода изменятся, чтобы отобразить количество выполняемой ядром работы. Вы заметите, что увеличилось системное время, больше данных перемещается

на диск и с него, а также больше процессов заблокировано, поскольку память, которую они намерены использовать, недоступна (она перемещена в область подкачки).

Я объяснил не все столбцы вывода команды `vmstat`. Узнать подробности вы можете на странице руководства vmstat(8). Чтобы лучше их понимать, сначала может потребоваться узнать больше о том, как ядро управляет памятью: из лекций или книги вроде *Operating System Concepts* («Общие представления об операционных системах»), 9-е издание (Wiley, 2012).

8.11. Отслеживание ввода/вывода

По умолчанию команда `vmstat` выводит некоторую общую статистику ввода/вывода. Хотя можно получить детализированные сведения об использовании ресурсов каждого раздела с помощью команды `vmstat -d`, в этом случае вывод будет довольно объемным. Попробуйте начать с инструмента, предназначенного только для статистики ввода/вывода, — команды `iostat`.

8.11.1. Использование команды iostat

Подобно команде `vmstat`, при запуске без параметров команда `iostat` показывает статистику за все время работы компьютера:

```
$ iostat
[kernel information]
avg-cpu:  %user  %nice %system %iowait  %steal   %idle
           4.46   0.01    0.67    0.31    0.00   94.55

Device:         tp s   kB_read/s   kB_wrtn/s   kB_read   kB_wrtn
sda            4.6 7       7.2 8       49.86   9493727  65011716
sde            0.0 0       0.0 0        0.00      1230         0
```

Часть `avg-cpu` в верхней части сообщает ту же информацию об использовании процессора, что и другие утилиты, которые вы видели в этой главе. Перейдите к нижней части, которая показывает для каждого из устройств следующее.

tps	Среднее количество пересылок данных в секунду
kB_read/s	Среднее количество считанных килобайтов в секунду
kB_wrtn/s	Среднее количество записанных килобайтов в секунду
kB_read	Общее количество считанных килобайтов
kB_wrtn	Общее количество записанных килобайтов

Еще одно сходство с командой `vmstat` таково: можно передавать величину интервала как аргумент, например `iostat 2`, чтобы результаты обновлялись каждые 2 секунды. При использовании интервала может потребоваться отобразить отчет только об устройстве. Для этого применяется параметр `-d` (например, `iostat -d 2`).

По умолчанию в отчете команды `iostat` не приводится информация о разделах. Чтобы отобразить всю такую информацию, используйте параметр `-p ALL`. Поскольку

в типичной системе бывает несколько разделов, вы получите обширный отчет. Вот фрагмент того, что вы можете увидеть:

```
$ iostat -p ALL
--snip
--Device:              tps       kB_read/s       kB_wrtn/s    kB_read     kB_wrtn
--snip-
sda                  4.67            7.27           49.83    9496139    65051472
sda1                 4.38            7.16           49.51    9352969    64635440
sda2                 0.00            0.00            0.00          6           0
sda5                 0.01            0.11            0.32     141884      416032
scd0                 0.00            0.00            0.00          0           0
--snip--
sde                  0.00            0.00            0.00       1230           0
```

В этом примере все устройства `sda1`, `sda2` и `sda5` являются разделами диска `sda`, поэтому между столбцами, относящимися к чтению и записи, будет небольшое наложение данных. Однако сумма значений, относящихся к разделам, не обязательно должна равняться значению для диска. Несмотря на то что чтение с устройства `sda1` также рассматривается как чтение с диска `sda`, помните о том, что с диска `sda` можно выполнять считывание напрямую, например при чтении таблицы разделов.

8.11.2. Отслеживание использования ввода/вывода каждого процесса с помощью команды iotop

Если вам необходимо копнуть глубже, чтобы увидеть ресурсы ввода/вывода, используемые отдельными процессами, вам может помочь команда `iotop`. Применение этой команды похоже на работу с командой `top`. Появляется постоянно обновляемый отчет, который показывает процессы, использующие большую часть ресурсов ввода/вывода, а общий итог приведен вверху:

```
# iotop
Total DISK READ:         4.76 K/s | Total DISK WRITE:     333.31 K/s
  TID  PRIO  USER       DISK READ DISK  WRITE  SWAPIN     IO> COMMAND
  260 be/3 root          0.00 B/s  38.09 K/s  0.00 %  6.98 % [jbd2/sda1-8]
 2611 be/4 juser         4.76 K/s  10.32 K/s  0.00 %  0.21 % zeitgeist-daemon
 2636 be/4 juser         0.00 B/s  84.12 K/s  0.00 %  0.20 % zeitgeist-fts
 1329 be/4 juser         0.00 B/s  65.87 K/s  0.00 %  0.03 % soffice.b~ash-pipe=6
 6845 be/4 juser 0.00 B/s 812.63 B/s 0.00 % 0.00 % chromium-browser
19069 be/4 juser 0.00 B/s 812.63 B/s 0.00 % 0.00 % rhythmbox
```

Обратите внимание на то, что здесь наряду со столбцами сведений о пользователе, команде и чтении/записи присутствует столбец `TID` (идентификатор потока) вместо идентификатора процесса. Инструмент `iotop` — одна из немногих утилит, которые отображают потоки вместо процессов.

Столбец `PRIO` (приоритет) отображает приоритет ввода/вывода. Он похож на приоритет процессора, который вы уже видели, но он влияет на то, насколько быстро ядро распределяет операции чтения и записи для процесса. В таком приори-

тете, как be/4, часть be является *классом обслуживания*, а число задает уровень приоритета. Как и для приоритетов процессора, более важными являются меньшие числа. Например, ядро отводит больше времени на ввод/вывод для процесса с приоритетом be/3, чем для процесса с приоритетом be/4.

Ядро использует класс обслуживания, чтобы обеспечить дополнительное управление планированием ввода/вывода. Вы увидите следующие три класса обслуживания в команде iotop.

- be — наилучший объем работы. Ядро старается наиболее справедливо распределить время ввода/вывода для этого класса. Большинство процессов запускаются в этом классе обслуживания.
- rt — реальное время. Ядро планирует любой ввод/вывод в реальном времени перед любым другим классом ввода/вывода, каким бы он ни был.
- idle — бездействие. Ядро выполняет ввод/вывод для этого класса только тогда, когда не должен быть выполнен никакой другой ввод/вывод. Для этого класса обслуживания не указывается уровень приоритета.

Можно проверить и изменить приоритет ввода/вывода для процесса с помощью утилиты ionice; подробности см. на странице руководства ionice(1). Хотя вам вряд ли потребуется беспокоиться о приоритетах ввода/вывода.

8.12. Отслеживание процессов с помощью команды pidstat

Вы увидели, как можно отслеживать конкретные процессы с помощью таких утилит, как top и iotop. Однако эти результаты обновляются в реальном времени, при каждом обновлении предыдущий отчет стирается. Утилита pidstat позволяет вам отследить использование ресурсов процессом с течением времени в стиле команды vmstat. Вот простой пример, в котором с ежесекундным обновлением отслеживается процесс 1329:

```
$ pidstat -p 1329 1
Linux 3.2.0-44-generic-pae (duplex)     07/01/2015      _i686_ (4 CPU)
09:26:55 PM       PID    %usr %system %guest %CPU CPU Command
09:27:03 PM      1329    8.00  0.00     0.00 8.00   1 myprocess
09:27:04 PM      1329    0.00  0.00     0.00 0.00   3 myprocess
09:27:05 PM      1329    3.00  0.00     0.00 3.00   1 myprocess
09:27:06 PM      1329    8.00  0.00     0.00 8.00   3 myprocess
09:27:07 PM      1329    2.00  0.00     0.00 2.00   3 myprocess
09:27:08 PM      1329    6.00  0.00     0.00 6.00   2 myprocess
```

В отчете по умолчанию приведены процентные отношения для пользовательского и системного времени, а также общая процентная доля процессорного времени. Есть даже сведения о том, на каком из процессоров запущен процесс. Столбец %guest представляет нечто необычное: это процентное отношение времени, которое процесс потратил на выполнение чего-либо внутри виртуальной машины. Если вы не запускаете виртуальную машину, не беспокойтесь о нем.

Хотя команда `pidstat` по умолчанию показывает использование процессора, она может намного больше этого. Например, можно применять параметр `-r`, чтобы отслеживать память, или параметр `-d`, чтобы включить отслеживание диска. Попробуйте применить их, а затем загляните на страницу руководства pidstat(1), чтобы узнать еще больше подробностей о потоках, переключении контекста или о чем-либо еще, что обсуждалось в данной главе.

8.13. Дополнительные темы

Одна из причин, почему существует так много инструментов для измерения использования ресурсов, в том, что множество типов ресурсов потребляется различными способами. В этой главе вы видели, как ресурсы процессора, памяти, ввода/вывода и системы использовались процессами, потоками внутри процессов и ядром.

Еще одна причина — ограниченность ресурсов. Чтобы система работала, ее компоненты должны стремиться к потреблению меньшего количества ресурсов. В прошлом за одним компьютером работало несколько пользователей, поэтому было необходимо обеспечить каждого из них достаточной долей ресурсов. Сейчас, хотя современные ПК могут и не иметь нескольких пользователей, они по-прежнему имеют множество процессов, соревнующихся за ресурсы. Точно так же и для высокопроизводительных сетевых серверов необходимо тщательное отслеживание ресурсов.

Дополнительные темы, относящиеся к отслеживанию ресурсов и анализу производительности, включают следующее.

- `sar` (System Activity Reporter, обозреватель системной активности). Пакет `sar` содержит многие из функций для непрерывного отслеживания команды `vmstat`, но он также выполняет запись использования ресурсов с течением времени. С помощью пакета `sar` можно узнать, что делала ваша система в определенный момент времени. Это удобно, когда необходимо проанализировать системное событие, которое уже произошло.
- `acct` (учет процессов). Пакет `acct` может регистрировать процессы и использование ресурсов ими.
- **Квоты.** Многие системные ресурсы можно ограничить в зависимости от процесса или от пользователя. Некоторые параметры применения процессора и памяти содержатся в файле `/etc/security/limits.conf`; есть также страница руководства limits.conf(5). Это функция стандарта PAM, и процессы будут подчиняться ей только тогда, когда они были запущены из чего-либо, что использует стандарт PAM (например, из оболочки входа в систему). Можно также ограничить количество дискового пространства, которое может потреблять пользователь, с помощью системы `quota`.

Если вы заинтересованы настройкой системы и, в частности, ее производительностью, книга Брендана Грегга (Brendan Gregg) *Systems Performance: Enterprise and the Cloud* («Производительность системы: предприятие и облако») (Prentice Hall, 2013) содержит намного больше подробностей.

Мы еще не коснулись множества инструментов, которые могут быть использованы при отслеживании потребления сетевых ресурсов. Чтобы их применять, сначала необходимо понять, как устроена сеть. Именно к этому мы сейчас и перейдем.