Санкт-Петербургский государственный политехнический университет
Институт Информационных Технологий и Управления
Кафедра компьютерных систем и программных технологий

Отчёт по практической работе
по предмету «Системное программное обеспечение»

Утилита tor

Работу выполнил студент гр. 53501/3 Мартынов С. А.

Работу принял преподаватель Душутина Е. В.

Санкт-Петербург
2015

Содержание

Постановка задачи

В рамках данной работы необходимо ознакомиться с работой утилиты top. Рассмотреть и описать механизмы сбора информации, используемые системные вызовы и общий порядок функционирования утилиты.

В работе необходимо указать источники и версии используемых программных продуктов.

Показать пример доступа к proc в утилите.

Привести отдельные собственные программные примеры доступа к proc (по чтению и записи посредством утилит и напрямую);

Предложить модификацию утилиты дополнением информации о входящих потоках (нитях) в заданный в качестве параметра процесс, а также по ключу информации о потоках каждого процесса.

Описать возможные способы получения и источники информации о потоках процесса, привести отдельные программные примеры (вне модифицированной утилиты) получения информации о входящих потоках для процесса.

Полные исходные коды сделать доступными по адресу `https://github.com/SemenMartynov/SPbPU_SystemProgramming`.

Введение

Работа выполняется под управлением Ubuntu 14.04.2 LTS. В качестве изучаемой утилиты используется реализация htop (автор Hisham H. Muhammad). Htop написан на языке Си и использует для отображения библиотеку Ncurses. Htop показывает динамический список системных процессов (рисунок 1), список обычно выравнивается по использованию ЦПУ. В отличие от top, htop показывает все процессы в системе. Также показывает время непрерывной работы, использование процессоров и памяти.

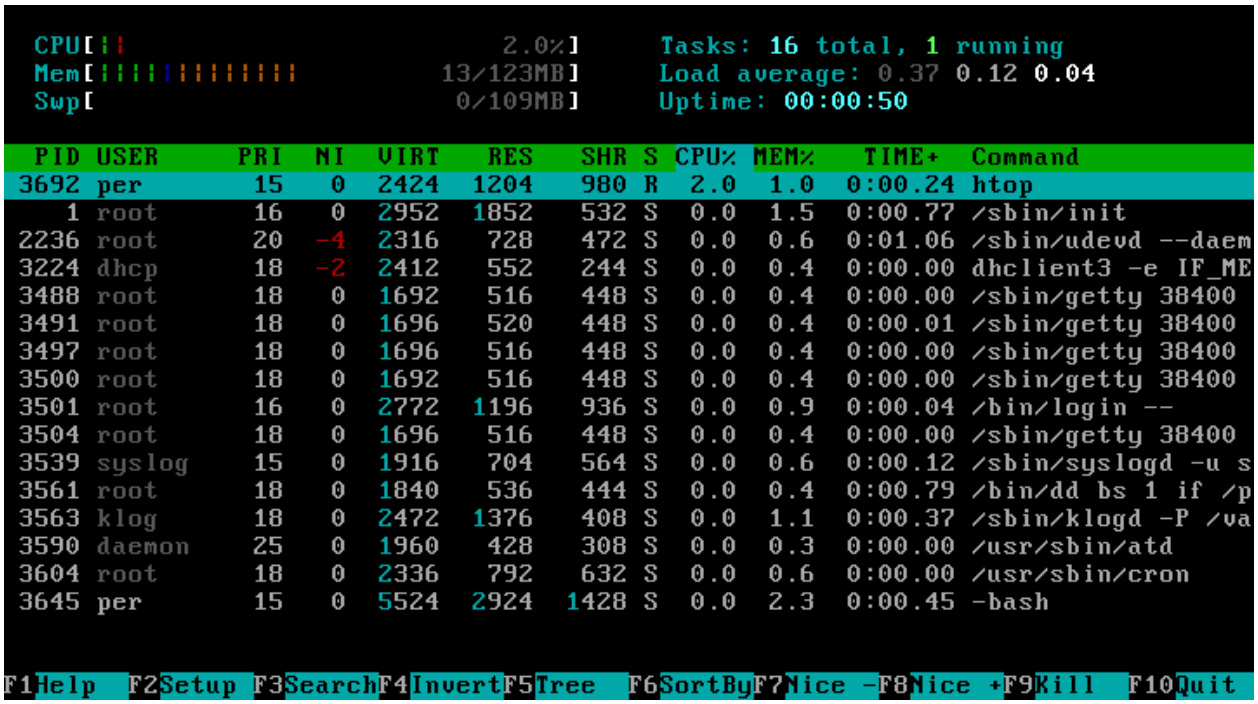

Рис. 1: Системный монитор htop

Рассматривается версия 1.0.3 (от 24 апреля 2014 года). Исходники доступны по лицензии GPL на сайте `http://hisham.hm/htop/`. Для простоты изучения, в код системы были внесены не значительные изменения.

1 Виртуальная файловая система procfs

Файловая система /proc содержит подробную информацию об активных процессах. Информация о процессе, сохраненная в файловой системе /proc, изменяется по мере прохождения данным процессом его жизненного цикла.

Каждый элемент в каталоге /proc - это десятичное число, соответствующее идентификатору какого-нибудь процесса. Каждый каталог в файловой системе /proc содержит файлы с более подробной информацией о данном процессе. Владелец каждого файла в каталоге /proc и его подкаталогах устанавливается по номеру идентификатора пользователя данного процесса.

Первоначально procfs была разработана для свободного получения информации о состоянии процессов, теперь ее функции расширились, и через эту виртуальную файловую систему процессам можно передавать какие-то параметры (см. рис. 2).

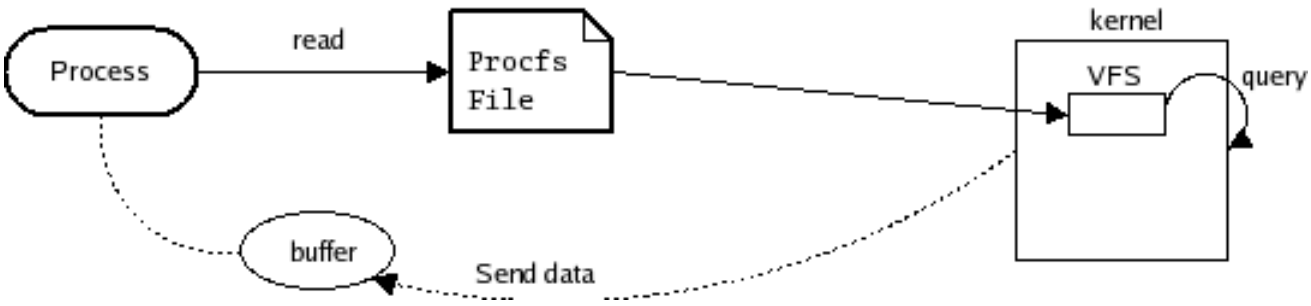

Рис. 2: Виртуальная файловая система procfs

Виртуальной она называется потому, что имеющиеся здесь файлы и каталоги на самом деле не находятся на жестком диске. После загрузки ядра они находятся в оперативной памяти. Для пользователя этот механизм полностью прозрачен. Многие программы, в том числе htop, собирают информацию из файлов в /proc, форматируют их и выводят результат.

В /proc можно найти также информацию об установленном оборудовании, разметке жесткого диска, статистику и многое другое.

При работе с /proc есть важный нюанс: информацию из файла можно прочитать, но открыть его в текстовом редакторе не получится, т.к. его содержимое может измениться в любую секунду. Для записи и считывания данных в такой файл используются утилиты, вроде cat и echo.

Файлы в procfs могут иметь три варианта доступа:

- только для чтения — предназначены для получения информации об определенном

параметре, при попытке в них что-то записать будет выдано предупреждение;

- только для чтения пользователем root — такой же, как и предыдущий, но получить информацию может только администратор;
- только для записи пользователем root — позволяет не только считать данные, но и изменить параметр.

Возможны и некоторые комбинации этих трех вариантов. К тому же в некоторые файлы можно записать только строго определенное значение.[1]

Утилита htop производит считывание различных параметров процессов. Далее будет рассмотрено как это происходит.

Просмотреть информацию о процессоре можно командой cat из файла /proc/cpuinfo. В моём случае это дало следующий результат (для отчёта вывод обрезан, т.к. повторяется по всем ядрам):

```
sam@spb:~$ cat /proc/cpuinfo
processor : 0
vendor_id : GenuineIntel
cpu family : 6
model : 23
model name : Intel(R) Core(TM)2 Quad CPU    Q8300  @ 2.50GHz
stepping : 10
microcode : 0xa0b
cpu MHz : 1998.000
cache size : 2048 KB
physical id : 0
siblings : 4
core id : 0
cpu cores : 4
apicid : 0
initial apicid : 0
fpu : yes
fpu_exception : yes
cpuid level : 13
wp : yes
flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat
 pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx lm constant_tsc
 arch_perfmon pebs bts rep_good nopl aperfmperf pni dtes64 monitor ds_cpl vmx est
```

```
 tm2 ssse3 cx16  xtpr pdcm sse4_1 xsave lahf_lm dtherm tpr_shadow vnmi flexpriority
bogomips : 4991.93
clflush size : 64
cache_alignment : 64
address sizes : 36 bits physical, 48 bits virtual
power management:
```

А вот вывод информации о памяти

```
sam@spb:~$ cat /proc/meminfo
MemTotal:        8176408 kB
MemFree:          391976 kB
Buffers:           39692 kB
Cached:           964508 kB
SwapCached:         1784 kB
Active:          1878368 kB
Inactive:        1060044 kB
Active(anon):    1528192 kB
Inactive(anon):   729880 kB
Active(file):     350176 kB
Inactive(file):   330164 kB
Unevictable:          32 kB
Mlocked:              32 kB
SwapTotal:       8387580 kB
SwapFree:        8359320 kB
Dirty:                 0 kB
Writeback:             0 kB
AnonPages:       1932372 kB
Mapped:          4824228 kB
Shmem:            323860 kB
Slab:             176288 kB
SReclaimable:     100712 kB
SUnreclaim:        75576 kB
KernelStack:        4568 kB
PageTables:        39424 kB
NFS_Unstable:          0 kB
Bounce:                0 kB
WritebackTmp:          0 kB
```

```
CommitLimit:    12475784 kB
Committed_AS:    9617760 kB
VmallocTotal:   34359738367 kB
VmallocUsed:      327844 kB
VmallocChunk:   34359407416 kB
HardwareCorrupted:     0 kB
AnonHugePages:    475136 kB
HugePages_Total:       0
HugePages_Free:        0
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
DirectMap4k:     1467968 kB
DirectMap2M:     6920192 kB
```

Следующий пример показывает как можно разрешить машине быть сетевым шлюзом для IPv6 соединений (параметр будет сброшен в 0 после перезагрузки):

```
sam@spb:~$ cat /proc/sys/net/ipv6/conf/all/forwarding
0
sam@spb:~$ echo "1" | sudo tee -a /proc/sys/net/ipv6/conf/all/forwarding
[sudo] password for sam:
1
sam@spb:~$ cat /proc/sys/net/ipv6/conf/all/forwarding
1
sam@spb:~$
```

2 Процессы

Листинг 1 содержит отрывок файла Process.h, описывающий структуру Process. На самом деле этот отрезок короче, если учесть работу препроцессора, который содержит многие строки (отладочную информацию, данные OpenVZ и виртуального сервера).

Листинг 1: Клиент именованного каналов (src/top/Process.h)

```
1  typedef struct Process_ {
2     Object super;
3
4     struct ProcessList_ *pl;
5     bool updated;
6
7     pid_t pid;
8     char* comm;
9     int indent;
10    char state;
11    bool tag;
12    bool showChildren;
13    bool show;
14    pid_t ppid;
15    unsigned int pgrp;
16    unsigned int session;
17    unsigned int tty_nr;
18    pid_t tgid;
19    int tpgid;
20    unsigned long int flags;
21    #ifdef DEBUG
22    unsigned long int minflt;
23    unsigned long int cminflt;
24    unsigned long int majflt;
25    unsigned long int cmajflt;
26    #endif
27    unsigned long long int utime;
28    unsigned long long int stime;
29    unsigned long long int cutime;
30    unsigned long long int cstime;
31    long int priority;
32    long int nice;
33    long int nlwp;
34    IOPriority ioPriority;
35    char starttime_show[8];
36    time_t starttime_ctime;
37    #ifdef DEBUG
```

```
38    long int itrealvalue;
39    unsigned long int vsize;
40    long int rss;
41    unsigned long int rlim;
42    unsigned long int startcode;
43    unsigned long int endcode;
44    unsigned long int startstack;
45    unsigned long int kstkesp;
46    unsigned long int kstkeip;
47    unsigned long int signal;
48    unsigned long int blocked;
49    unsigned long int sigignore;
50    unsigned long int sigcatch;
51    unsigned long int wchan;
52    unsigned long int nswap;
53    unsigned long int cnswap;
54    #endif
55    int exit_signal;
56    int processor;
57    int m_size;
58    int m_resident;
59    int m_share;
60    int m_trs;
61    int m_drs;
62    int m_lrs;
63    int m_dt;
64    uid_t st_uid;
65    float percent_cpu;
66    float percent_mem;
67    char* user;
68    #ifdef HAVE_OPENVZ
69    unsigned int ctid;
70    unsigned int vpid;
71    #endif
72    #ifdef HAVE_VSERVER
73    unsigned int vxid;
74    #endif
75    #ifdef HAVE_TASKSTATS
76    unsigned long long io_rchar;
77    unsigned long long io_wchar;
78    unsigned long long io_syscr;
79    unsigned long long io_syscw;
80    unsigned long long io_read_bytes;
81    unsigned long long io_write_bytes;
82    unsigned long long io_cancelled_write_bytes;
```

```
83    double io_rate_read_bps;
84    unsigned long long io_rate_read_time;
85    double io_rate_write_bps;
86    unsigned long long io_rate_write_time;
87    #endif
88    #ifdef HAVE_CGROUP
89    char* cgroup;
90    #endif
91 } Process;
```

Назначение большинства полей понятно исходя из их названий, и они здесь представлены для дальнейшего перехода к системам, которые заполняют значение этих полей (к примеру, процент занятой памяти или процессорного времени). Процессы объединяются в списки (строка 4), а объект Object (строка 2) отвечает за отображение. Полный граф взаимодействия структура данных представлен на рисунке 3.

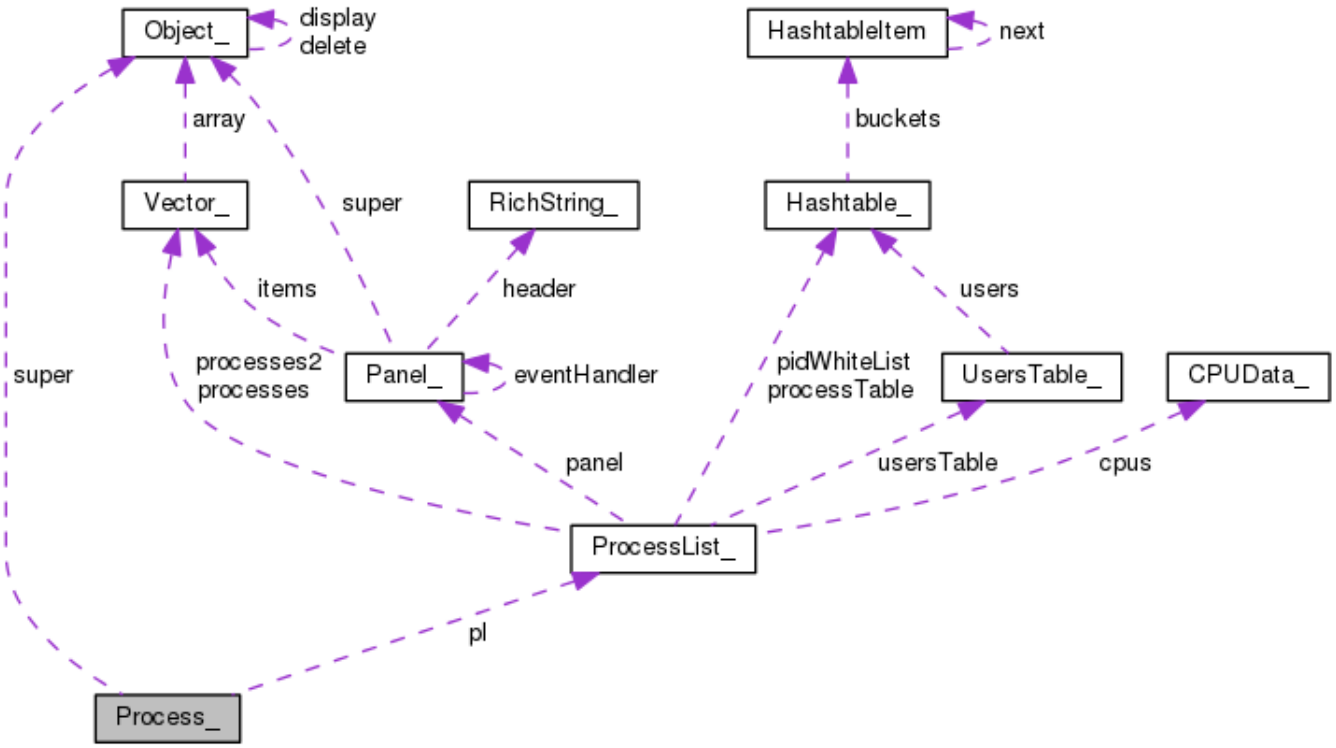

Рис. 3: Граф взаимодействия для структуры Process

На графе видно, что некоторые структуры, такие как CPUData_ не связаны с конкретным процессом, и это логично, т.к. эта общесистемная информация, но к ней есть доступ через сам список. Структуры, такие как RitcString_, Vector_ и Hashtable_ используются для служебных целей (сортировка списков, парсинг текстовой информации), т.к. программа написана на C (т.е. это реализация структур из C++, реализованная на языке C).

Все процессы помещаются в список, который, помимо самих процессоров, хранит ещё общесистемные показатели, такие как распределение памяти и ресурсов центрального процессора (листинг 2).

Получить показатели используемой памяти можно из файла /proc/meminfo (стр. 5 - макрос подставит правильный путь на место PROCMEMINFOFILE). В этой функции определяется следующие показатели памяти:

- общий объём памяти (стр 17);
- объём свободной памяти (стр. 19);
- объём памяти, разделяемой между процессами (стр. 21);
- размер буферов (стр. 25);
- размер кешей (стр.29);
- общий объём, используемый для хранения страниц памяти на диске (стр. 33);
- свободное место в свапе (стр. 35);
- объём используемой памяти, как разность между общим объёмом и свободной памятью (стр. 41);
- объём используемого места в свапе, как разность между общим объёмом и свободным пространством (стр. 42);

Помимо информации о памяти, тут же происходит считывание информации о прерываниях и операциях ввода-вывода из файла /proc/stat (стр. 45 - замена имени файла будет произведена препроцессором). из этого файла для каждого процессора (стр. 49) считываются следующие показатели:

- время, проведённое процессором в пространстве пользователя (стр. 83);
- время, на выделение которого повлиял приоритет процесса (стр. 84);
- время, проведённое процессором в пространстве ядра (стр. 85);
- время, проведённое процессором в пространстве ядра, включая обработку прерываний (стр. 86);
- время, проведённое процессором в режиме бездействия (стр. 87);
- время, проведённое процессором в режиме бездействия, включая время ожидания выполнения операций ввода-вывода (стр. 88);
- время ожидания выполнения операций ввода-вывода (стр. 89);

- объём обработки прерываний (стр. 90);
- объём обработки программных прерываний (стр. 91);
- украденное время - характерно для гипервизоров (стр. 92);
- время в режиме гостя - характерно для гипервизоров (стр. 93);
- общее время работы процессора (стр. 94);
- объём используемого места в свапе, как разность между общим объёмом и свободным пространством (стр. 42);

Листинг 2: Считывание различных общесистемных показателей(src/top/ProcessList.c)

```
1  void ProcessList_scan(ProcessList* this) {
2     unsigned long long int usertime, nicetime, systemtime, systemalltime,
          idlealltime, idletime, totaltime, virtalltime;
3     unsigned long long int swapFree = 0;
4
5     FILE* file = fopen(PROCMEMINFOFILE, "r");
6     if (file == NULL) {
7        CRT_fatalError("Cannot open " PROCMEMINFOFILE);
8     }
9     int cpus = this->cpuCount;
10    {
11       char buffer[128];
12       while (fgets(buffer, 128, file)) {
13
14          switch (buffer[0]) {
15          case 'M':
16             if (String_startsWith(buffer, "MemTotal:"))
17                sscanf(buffer, "MemTotal: %llu kB", &this->totalMem);
18             else if (String_startsWith(buffer, "MemFree:"))
19                sscanf(buffer, "MemFree: %llu kB", &this->freeMem);
20             else if (String_startsWith(buffer, "MemShared:"))
21                sscanf(buffer, "MemShared: %llu kB", &this->sharedMem);
22             break;
23          case 'B':
24             if (String_startsWith(buffer, "Buffers:"))
25                sscanf(buffer, "Buffers: %llu kB", &this->buffersMem);
26             break;
27          case 'C':
28             if (String_startsWith(buffer, "Cached:"))
29                sscanf(buffer, "Cached: %llu kB", &this->cachedMem);
30             break;
31          case 'S':
```

```
32              if (String_startsWith(buffer, "SwapTotal:"))
33                 sscanf(buffer, "SwapTotal: %llu kB", &this->totalSwap);
34              if (String_startsWith(buffer, "SwapFree:"))
35                 sscanf(buffer, "SwapFree: %llu kB", &swapFree);
36              break;
37           }
38        }
39     }
40
41     this->usedMem = this->totalMem - this->freeMem;
42     this->usedSwap = this->totalSwap - swapFree;
43     fclose(file);
44
45     file = fopen(PROCSTATFILE, "r");
46     if (file == NULL) {
47        CRT_fatalError("Cannot open " PROCSTATFILE);
48     }
49     for (int i = 0; i <= cpus; i++) {
50        char buffer[256];
51        int cpuid;
52        unsigned long long int ioWait, irq, softIrq, steal, guest;
53        ioWait = irq = softIrq = steal = guest = 0;
54        // Dependending on your kernel version,
55        // 5, 7 or 8 of these fields will be set.
56        // The rest will remain at zero.
57        fgets(buffer, 255, file);
58        if (i == 0)
59           sscanf(buffer, "cpu  %llu %llu %llu %llu %llu %llu %llu %llu %llu",
                &usertime, &nicetime, &systemtime, &idletime, &ioWait, &irq, &
               softIrq, &steal, &guest);
60        else {
61           sscanf(buffer, "cpu%d %llu %llu %llu %llu %llu %llu %llu %llu %llu"
               , &cpuid, &usertime, &nicetime, &systemtime, &idletime, &ioWait,
                &irq, &softIrq, &steal, &guest);
62           //assert(cpuid == i - 1);
63        }
64        // Fields existing on kernels >= 2.6
65        // (and RHEL's patched kernel 2.4...)
66        idlealltime = idletime + ioWait;
67        systemalltime = systemtime + irq + softIrq;
68        virtalltime = steal + guest;
69        totaltime = usertime + nicetime + systemalltime + idlealltime +
              virtalltime;
70        CPUData* cpuData = &(this->cpus[i]);
71        //assert (usertime >= cpuData->userTime);
```

```
72          //assert (nicetime >= cpuData->niceTime);
73          //assert (systemtime >= cpuData->systemTime);
74          //assert (idletime >= cpuData->idleTime);
75          //assert (totaltime >= cpuData->totalTime);
76          //assert (systemalltime >= cpuData->systemAllTime);
77          //assert (idlealltime >= cpuData->idleAllTime);
78          //assert (ioWait >= cpuData->ioWaitTime);
79          //assert (irq >= cpuData->irqTime);
80          //assert (softIrq >= cpuData->softIrqTime);
81          //assert (steal >= cpuData->stealTime);
82          //assert (guest >= cpuData->guestTime);
83          cpuData->userPeriod = usertime - cpuData->userTime;
84          cpuData->nicePeriod = nicetime - cpuData->niceTime;
85          cpuData->systemPeriod = systemtime - cpuData->systemTime;
86          cpuData->systemAllPeriod = systemalltime - cpuData->systemAllTime;
87          cpuData->idleAllPeriod = idlealltime - cpuData->idleAllTime;
88          cpuData->idlePeriod = idletime - cpuData->idleTime;
89          cpuData->ioWaitPeriod = ioWait - cpuData->ioWaitTime;
90          cpuData->irqPeriod = irq - cpuData->irqTime;
91          cpuData->softIrqPeriod = softIrq - cpuData->softIrqTime;
92          cpuData->stealPeriod = steal - cpuData->stealTime;
93          cpuData->guestPeriod = guest - cpuData->guestTime;
94          cpuData->totalPeriod = totaltime - cpuData->totalTime;
95          cpuData->userTime = usertime;
96          cpuData->niceTime = nicetime;
97          cpuData->systemTime = systemtime;
98          cpuData->systemAllTime = systemalltime;
99          cpuData->idleAllTime = idlealltime;
100         cpuData->idleTime = idletime;
101         cpuData->ioWaitTime = ioWait;
102         cpuData->irqTime = irq;
103         cpuData->softIrqTime = softIrq;
104         cpuData->stealTime = steal;
105         cpuData->guestTime = guest;
106         cpuData->totalTime = totaltime;
107      }
108      double period = (double)this->cpus[0].totalPeriod / cpus; fclose(file);
109
110      // mark all process as "dirty"
111      for (int i = 0; i < Vector_size(this->processes); i++) {
112         Process* p = (Process*) Vector_get(this->processes, i);
113         p->updated = false;
114      }
115
116      this->totalTasks = 0;
```

```
117    this->userlandThreads = 0;
118    this->kernelThreads = 0;
119    this->runningTasks = 0;
120
121    ProcessList_processEntries(this, PROCDIR, NULL, period);
122
123    this->showingThreadNames = this->showThreadNames;
124
125    for (int i = Vector_size(this->processes) - 1; i >= 0; i--) {
126       Process* p = (Process*) Vector_get(this->processes, i);
127       if (p->updated == false)
128          ProcessList_remove(this, p);
129       else
130          p->updated = false;
131    }
132
133 }
```

3 Измерение уровня заряда батарейки

Листинг 3 содержит отрывок файла BatteryMeter.c, который содержит функционал для отображения информации об источнике питания.

Как видно в строке 18, информация о батарее находиться по пути /proc/acpi/battery. В этой директории должны находиться файлы, название которых начинается с букв BAT, и это сравнение производится в строке 38. Имена файлов складываются в односвязный список myList (стр. 44), для последующего перебора (стр. 48). В строке 53 формируется полный путь до файла, отображающего состояние батарейки и содержимое этого файла вычитывается в массив line (стр. 61). В этой строке производится смещение на количество символов, переданное в качестве параметра функции, а полученный результат провидится к числу и аккумулируется. Так собирается информация об общем объёме батареи и её состоянии (остатке заряда). Сбор осуществляет функция getProcBatData в стр. 186. Эта же функция определяет процент использования аккумулятора, путём деления остатка заряда на общий объём батареи (стр. 195).

Если источником питания является подключенный сетевой адаптер, то информация об этом может быть получена из /proc/acpi/ac_adapter (стр. 84) или из /sys/class/power_supply/ (стр. 132). В обоих случаях производится попытка чтения файла для определения активности адаптера. Этот вариант применим для ноутбуков, в моём случае обе директории оказались пусты.

В этом же файле представлен функционал для отслеживания подключения адаптера (в этот момент источник питания изменяется с батарейки на питание от электросети) или завершение зарядки аккумулятора. События мониторятся по пути /sys/class/power_supply/ (стр. 220). Код в строке 228 и строке 240 во многом дублируются, т.к. в некоторых системах событие окончание зарядки содержит слово CHARGE, а в некоторых ENERGY. Обработка события происходит в функции из строки 1.

Источник питания может быть электросетью, а может быть аккумуляторной батареей, выводимый текст определяется в строке 275. Полный граф включения для файла BatteryMeter.c представлен на рисунке 4.

Листинг 3: Battery Meter - измерение уровня заряда(src/top/BatteryMeter.c)

```
1  static unsigned long int parseUevent(FILE * file, const char *key) {
2     char line[100];
3     unsigned long int dValue = 0;
4
5     while (fgets(line, sizeof line, file)) {
6        if (strncmp(line, key, strlen(key)) == 0) {
7           char *value;
8           strtok(line, "=");
9           value = strtok(NULL, "=");
10          dValue = atoi(value);
11          break;
12       }
13    }
14    return dValue;
15 }
16
17 static unsigned long int parseBatInfo(const char *fileName, const unsigned
      short int lineNum, const unsigned short int wordNum) {
18    const char batteryPath[] = PROCDIR "/acpi/battery/";
19    DIR* batteryDir = opendir(batteryPath);
20    if (!batteryDir)
21       return 0;
22
23    typedef struct listLbl {
24       char *content;
25       struct listLbl *next;
26    } list;
27
28    list *myList = NULL;
29    list *newEntry;
30
31    /*
32       Some of this is based off of code found in kismet (they claim it came
          from gkrellm).
33       Written for multi battery use...
34     */
35    for (const struct dirent* dirEntries = readdir((DIR *) batteryDir);
         dirEntries; dirEntries = readdir((DIR *) batteryDir)) {
36       char* entryName = (char *) dirEntries->d_name;
37
38       if (strncmp(entryName, "BAT", 3))
39          continue;
40
41       newEntry = calloc(1, sizeof(list));
```

```
42          newEntry->next = myList;
43          newEntry->content = entryName;
44          myList = newEntry;
45      }
46
47      unsigned long int total = 0;
48      for (newEntry = myList; newEntry; newEntry = newEntry->next) {
49          const char infoPath[30];
50          const FILE *file;
51          char line[50];
52
53          snprintf((char *) infoPath, sizeof infoPath, "%s%s/%s", batteryPath,
                newEntry->content, fileName);
54
55          if ((file = fopen(infoPath, "r")) == NULL) {
56              closedir(batteryDir);
57              return 0;
58          }
59
60          for (unsigned short int i = 0; i < lineNum; i++) {
61              fgets(line, sizeof line, (FILE *) file);
62          }
63
64          fclose((FILE *) file);
65
66          const char *foundNumTmp = String_getToken(line, wordNum);
67          const unsigned long int foundNum = atoi(foundNumTmp);
68          free((char *) foundNumTmp);
69
70          total += foundNum;
71      }
72
73      free(myList);
74      free(newEntry);
75      closedir(batteryDir);
76      return total;
77  }
78
79  static ACPresence chkIsOnline() {
80      FILE *file = NULL;
81      ACPresence isOn = AC_ERROR;
82
83      if (access(PROCDIR "/acpi/ac_adapter", F_OK) == 0) {
84          const char *power_supplyPath = PROCDIR "/acpi/ac_adapter";
85          DIR *power_supplyDir = opendir(power_supplyPath);
```

```
86      if (!power_supplyDir)
87         return AC_ERROR;
88
89      for (const struct dirent *dirEntries = readdir((DIR *) power_supplyDir
           ); dirEntries; dirEntries = readdir((DIR *) power_supplyDir)) {
90         char* entryName = (char *) dirEntries->d_name;
91
92         if (entryName[0] != 'A')
93            continue;
94
95         char statePath[50];
96         snprintf((char *) statePath, sizeof statePath, "%s/%s/state",
              power_supplyPath, entryName);
97         file = fopen(statePath, "r");
98
99         if (!file) {
100           isOn = AC_ERROR;
101           continue;
102        }
103
104        char line[100];
105        fgets(line, sizeof line, file);
106        line[sizeof(line) - 1] = '\0';
107
108        if (file) {
109           fclose(file);
110           file = NULL;
111        }
112
113        const char *isOnline = String_getToken(line, 2);
114
115        if (strcmp(isOnline, "on-line") == 0) {
116           free((char *) isOnline);
117           isOn = AC_PRESENT;
118           // If any AC adapter is being used then stop
119           break;
120
121        } else {
122           isOn = AC_ABSENT;
123        }
124        free((char *) isOnline);
125     }
126
127     if (power_supplyDir)
128        closedir(power_supplyDir);
```

```
129
130     } else {
131
132         const char *power_supplyPath = "/sys/class/power_supply";
133
134         if (access("/sys/class/power_supply", F_OK) == 0) {
135             const struct dirent *dirEntries;
136             DIR *power_supplyDir = opendir(power_supplyPath);
137             char *entryName;
138
139             if (!power_supplyDir) {
140                 return AC_ERROR;
141             }
142
143             for (dirEntries = readdir((DIR *) power_supplyDir); dirEntries;
                    dirEntries = readdir((DIR *) power_supplyDir)) {
144                 entryName = (char *) dirEntries->d_name;
145
146                 if (strncmp(entryName, "A", 1)) {
147                     continue;
148                 }
149
150                 char onlinePath[50];
151                 snprintf((char *) onlinePath, sizeof onlinePath, "%s/%s/online",
                        power_supplyPath, entryName);
152                 file = fopen(onlinePath, "r");
153
154                 if (!file) {
155                     isOn = AC_ERROR;
156                     continue;
157                 }
158
159                 isOn = (fgetc(file) - '0');
160
161                 if (file) {
162                     fclose(file);
163                     file = NULL;
164                 }
165
166                 if (isOn == AC_PRESENT) {
167                     // If any AC adapter is being used then stop
168                     break;
169                 } else {
170                     continue;
171                 }
```

```
172             }
173
174             if (power_supplyDir)
175                 closedir(power_supplyDir);
176         }
177     }
178
179     // Just in case :-)
180     if (file)
181         fclose(file);
182
183     return isOn;
184 }
185
186 static double getProcBatData() {
187     const unsigned long int totalFull = parseBatInfo("info", 3, 4);
188     if (totalFull == 0)
189         return 0;
190
191     const unsigned long int totalRemain = parseBatInfo("state", 5, 3);
192     if (totalRemain == 0)
193         return 0;
194
195     double percent = totalFull > 0 ? ((double) totalRemain * 100) / (double)
            totalFull : 0;
196     return percent;
197 }
198
199 static double getSysBatData() {
200     const struct dirent *dirEntries;
201     const char *power_supplyPath = "/sys/class/power_supply/";
202     DIR *power_supplyDir = opendir(power_supplyPath);
203     if (!power_supplyDir)
204         return 0;
205
206     char *entryName;
207
208     unsigned long int totalFull = 0;
209     unsigned long int totalRemain = 0;
210
211     for (dirEntries = readdir((DIR *) power_supplyDir); dirEntries;
            dirEntries = readdir((DIR *) power_supplyDir)) {
212         entryName = (char *) dirEntries->d_name;
213
214         if (strncmp(entryName, "BAT", 3)) {
```

```
215             continue;
216         }
217 
218         const char ueventPath[50];
219 
220         snprintf((char *) ueventPath, sizeof ueventPath, "%s%s/uevent",
                power_supplyPath, entryName);
221 
222         FILE *file;
223         if ((file = fopen(ueventPath, "r")) == NULL) {
224             closedir(power_supplyDir);
225             return 0;
226         }
227 
228         if ((totalFull += parseUevent(file, "POWER_SUPPLY_ENERGY_FULL="))) {
229             totalRemain += parseUevent(file, "POWER_SUPPLY_ENERGY_NOW=");
230         } else {
231             //reset file pointer
232             if (fseek(file, 0, SEEK_SET) < 0) {
233                 closedir(power_supplyDir);
234                 fclose(file);
235                 return 0;
236             }
237         }
238 
239         //Some systems have it as CHARGE instead of ENERGY.
240         if ((totalFull += parseUevent(file, "POWER_SUPPLY_CHARGE_FULL="))) {
241             totalRemain += parseUevent(file, "POWER_SUPPLY_CHARGE_NOW=");
242         } else {
243           //reset file pointer
244             if (fseek(file, 0, SEEK_SET) < 0) {
245                 closedir(power_supplyDir);
246                 fclose(file);
247                 return 0;
248             }
249         }
250 
251         fclose(file);
252     }
253 
254     const double percent = totalFull > 0 ? ((double) totalRemain * 100) / (
            double) totalFull : 0;
255     closedir(power_supplyDir);
256     return percent;
257 }
```

```
258
259 static void BatteryMeter_setValues(Meter * this, char *buffer, int len) {
260    double percent = getProcBatData();
```

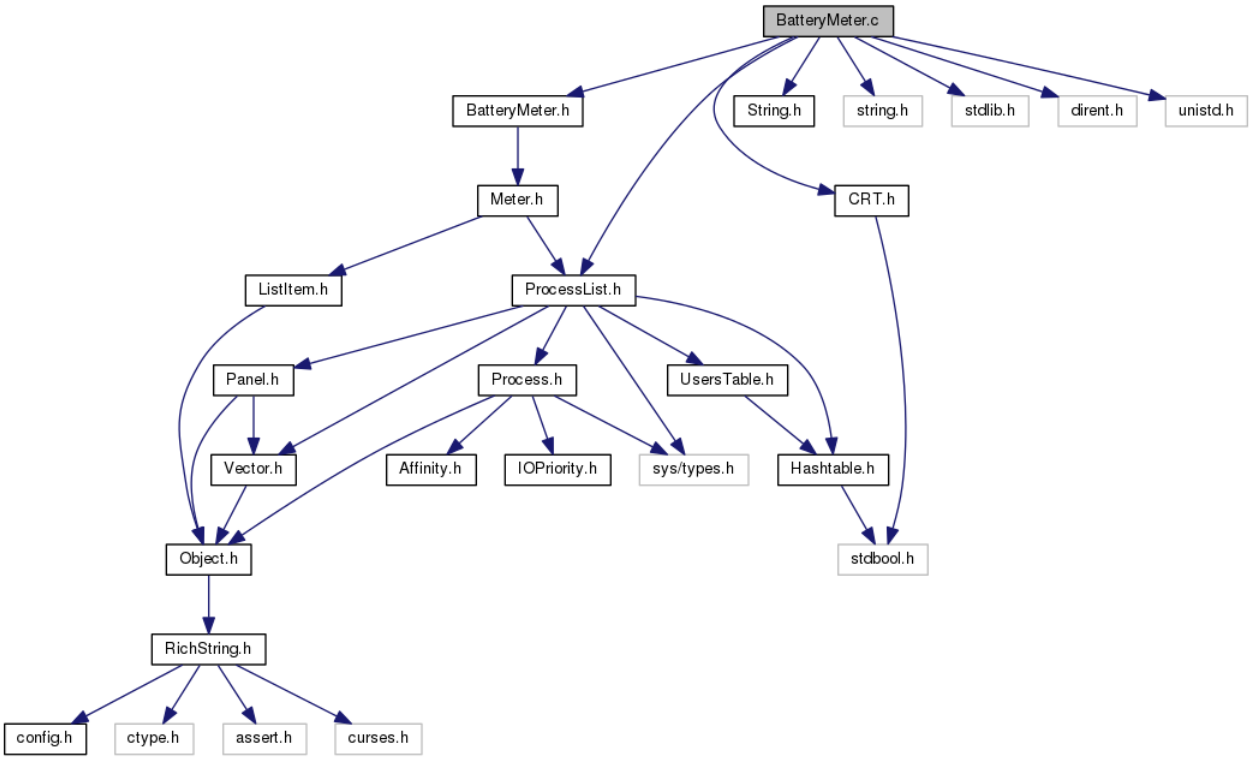

Рис. 4: Граф включения для файла BatteryMeter.c

4 Мониторинг времени

Монитор времени отвечает за отображение текущего времени пользователя. Отрывок файла ClockMeter.c представлен в листинге 4 а граф включения на рисунке 5.

Интерес представляет только одна функция. Она берёт локальное время (стр. 3) и преобразует его к привычному виду (часы:минуты:секунды), сохраняя это преобразование в буфере, который является аргументом функции.

Листинг 4: Clock Meter - мониторинг времени(src/top/ClockMeter.c)

```
1 static void ClockMeter_setValues(Meter* this, char* buffer, int size) {
2    time_t t = time(NULL);
3    struct tm *lt = localtime(&t);
4    this->values[0] = lt->tm_hour * 60 + lt->tm_min;
5    strftime(buffer, size, "%H:%M:%S", lt);
6 }
```

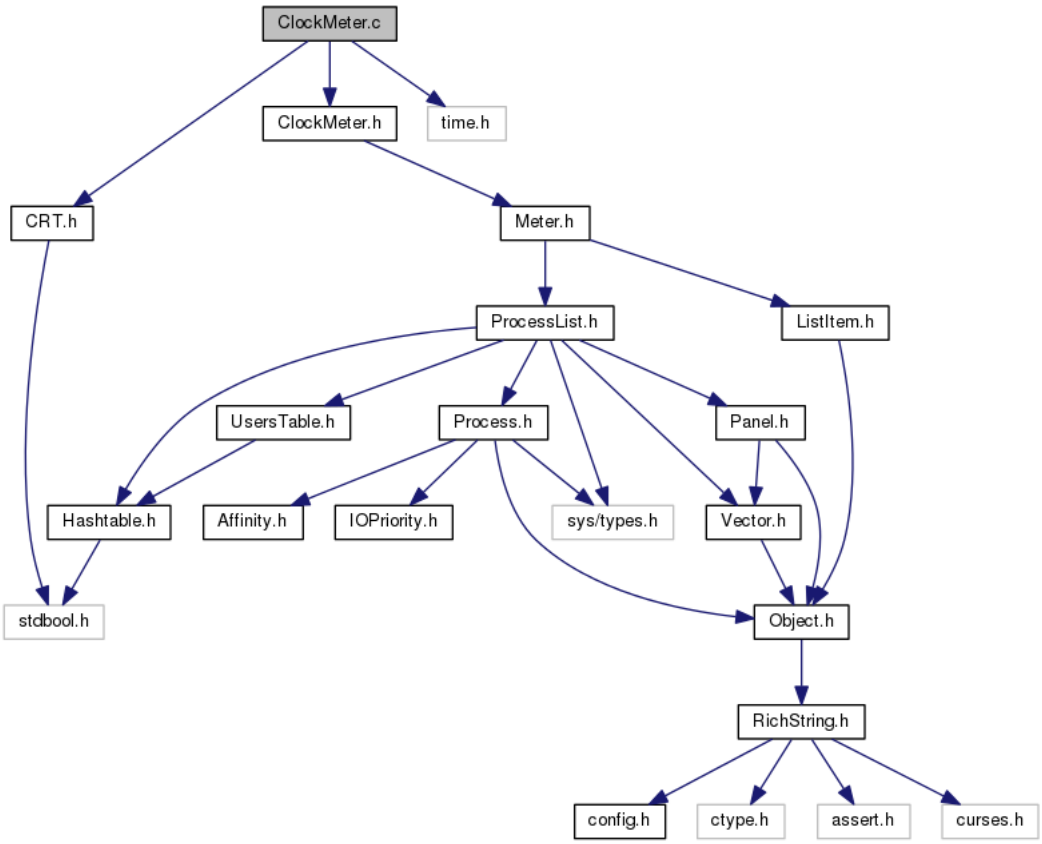

Рис. 5: Граф включения для файла ClockMeter.c

5 Центральный процессор

С оценкой потребления процессора всегда встает сложность разделения физических и логических ядер. В одном реальном процессоре может быть несколько ядер, каждое из которых может иметь дополнительный набор регистров, обеспечивающих hyper-threading.

Ранее мы рассматривали как происходит чтение информации об использовании процессора приложениями. Функция CPUMeter_setValues из листинга 5 занимается подготовкой следующих полей[2]:

- nicePeriod (стр. 11) - означает процент CPU, используемого пользовательскими процессами, на которые повлияло использование команд nice или renice, т.е. по существу их приоритет был изменен по сравнению с приоритетом по умолчанию, назначаемому планировщиком, на более высокий или низкий. При назначении какому-либо процессу команды nice, положительное число означает более низкий приоритет (1 = 1 шаг ниже нормального), а отрицательное число означает более высокий приоритет. 0 – значение по умолчанию, что означает, что решение о приоритете принимает планировщик. Можно установить, какой планировщик используется системой.
- userPeriod (стр. 12) - оказывает использование отдельного процессора (пользовательскими процессами, такими, как apache, mysql и т.д.) до максимального значения, составляющего 100%. Таким образом, если в четырехъядерном процессоре 1 процесс использует 100% CPU, это даст значение %us, равное 25%. Значение 12,5% для 8-ядерного процессора означает, что занято одно ядро.
- systemPeriod (стр 14) - означает использование CPU системой. Обычно это значение невысоко, высокие его значения могут свидетельствовать о проблеме с конфигами ядра, проблему со стороны драйвера, или целый ряд других вещей.
- irqPeriod (стр 15) - означает прерывания на уровне железа; на плате электроны движутся по микросхемам предсказуемым образом. Например, когда сетевая карта получает пакет, перед передачей информации, содержащейся в пакете в процессор через ядро, она запросит прерывание в канале прерывания материнской платы. Процессор сообщает ядру, что у сетевой карты для него есть информация, а ядро имеет возможность решить, как поступить. Высокое значение времени, тратящегося на обработку прерываний на уровне железа встречается на виртуальной машине довольно редко, но по мере того, как гипервизоры предоставляют в распоряжение виртуальных машин все больше «железа», эта ситуация может измениться. Чрезвычайно высокая пропускная способность сети, использование USB, вычисления на графических процессорах, – все это может привести к росту этого параметра на

величину, превышающую несколько процентов.

- softIrqPeriod (стр 16) - прерывание на уровне софта; начиная с ядра linux версии 2.4 реализована возможность запроса прерывания программным обеспечением (приложениями), а не элементом аппаратного обеспечения или устройством (драйвером), запрашивающим прерывание в канале прерывания материнской платы; запрос обслуживается ядром посредством его обработчика прерываний. Это означает, что приложение может запросить приоритетный статус, ядро может подтвердить получение команда, а программное обеспечение будет терпеливо ждать, пока прерывание не будет обслужено. Если мы применим утилиту tcpdump к гигабитному каналу с высоким трафиком, то значение может измениться примерно на 10%, – по мере заполнения выделенной памяти tcpdump, утилита посылает зарос на прерывание, чтобы переместить данные со стека на диск, экран и т.д.

- ioWaitPeriod (стр 17) - процент времени (циклов, секунд), в течение которого процессор простаивал, ожидая завершения операции ввода-вывода. Когда какой-либо процесс или программа запрашивает данные, он сначала проверяет кэш процессора (в нем имеется 2 или 3 кэша), затем проверяет память и, наконец, доходит до диска. Дойдя до диска, процессу или программе обычно приходится ждать, пока поток ввода-вывода передаст информацию в оперативную память, прежде чем иметь возможность снова на нем работать. Чем медленнее диск, тем выше будет значение IO Wait % для каждого процесса. Это происходит также с процессами записи на диск, если системный буфер заполнен и его необходимо прочистить при помощи ядра – обычно это наблюдается на серверах баз данных с высокой нагрузкой. Если значение IO Wait стабильно превышает 100 / (кол-во CPU * кол-во процессов)%, это означает, что, возможно, имеется проблема хранения, с которой необходимо разобраться. Если вы наблюдаете высокую среднюю нагрузку, прежде всего, проверьте этот параметр. Если он высок, тогда узкое место в процессах, скапливающихся на диске, а не в чем-либо еще.

- stealPeriod (стр 18) - в виртуализированной среде множество логических серверов могут работать под одним фактическим гипервизором. Каждой виртуальной машине(VM) присваивается 4-8 "виртуальных"CPU; хотя сами гипервизоры могут не иметь (кол-во VM * кол-во виртуальных CPU на одну VM). Причина этого заключается в том, что мы не перегружаем CPU использованием виртуальных машин, так что если дать одной-двум VM возможность изредка использовать 8 процессоров, это не будет негативно влиять на весь пул в целом. Однако если виртуальными процессорами VM используется количество CPU, превышающее количество физических (или логических, в случае с гиперпотоковыми процессорами Xeon), тогда значение iosteal

будет расти.

- guestPeriod (стр 19) - параметр похож на предыдущий, но со стороны виртуальной машины.

Функция CPUMeter_display (стр. 33) занимается оформленным выводом информации, которую мы описали выше. Граф включения представлен на рисунке 6.

Листинг 5: CPU Meter - центральный процессор(src/top/CPUMeter.c)

```
1  static void CPUMeter_setValues(Meter* this, char* buffer, int size) {
2     ProcessList* pl = this->pl;
3     int cpu = this->param;
4     if (cpu > this->pl->cpuCount) {
5        snprintf(buffer, size, "absent");
6        return;
7     }
8     CPUData* cpuData = &(pl->cpus[cpu]);
9     double total = (double) ( cpuData->totalPeriod == 0 ? 1 : cpuData->
         totalPeriod);
10    double percent;
11    this->values[0] = cpuData->nicePeriod / total * 100.0;
12    this->values[1] = cpuData->userPeriod / total * 100.0;
13    if (pl->detailedCPUTime) {
14       this->values[2] = cpuData->systemPeriod / total * 100.0;
15       this->values[3] = cpuData->irqPeriod / total * 100.0;
16       this->values[4] = cpuData->softIrqPeriod / total * 100.0;
17       this->values[5] = cpuData->ioWaitPeriod / total * 100.0;
18       this->values[6] = cpuData->stealPeriod / total * 100.0;
19       this->values[7] = cpuData->guestPeriod / total * 100.0;
20       this->type->items = 8;
21       percent = MIN(100.0, MAX(0.0, (this->values[0]+this->values[1]+this->
            values[2]+
22                        this->values[3]+this->values[4])));
23    } else {
24       this->values[2] = cpuData->systemAllPeriod / total * 100.0;
25       this->values[3] = (cpuData->stealPeriod + cpuData->guestPeriod) /
            total * 100.0;
26       this->type->items = 4;
27       percent = MIN(100.0, MAX(0.0, (this->values[0]+this->values[1]+this->
            values[2]+this->values[3])));
28    }
29    if (isnan(percent)) percent = 0.0;
30    snprintf(buffer, size, "%5.1f%%", percent);
31 }
32
```

```
33 static void CPUMeter_display(Object* cast, RichString* out) {
34    char buffer[50];
35    Meter* this = (Meter*)cast;
36    RichString_prune(out);
37    if (this->param > this->pl->cpuCount) {
38       RichString_append(out, CRT_colors[METER_TEXT], "absent");
39       return;
40    }
41    sprintf(buffer, "%5.1f%% ", this->values[1]);
42    RichString_append(out, CRT_colors[METER_TEXT], ":");
43    RichString_append(out, CRT_colors[CPU_NORMAL], buffer);
44    if (this->pl->detailedCPUTime) {
45       sprintf(buffer, "%5.1f%% ", this->values[2]);
46       RichString_append(out, CRT_colors[METER_TEXT], "sy:");
47       RichString_append(out, CRT_colors[CPU_KERNEL], buffer);
48       sprintf(buffer, "%5.1f%% ", this->values[0]);
49       RichString_append(out, CRT_colors[METER_TEXT], "ni:");
50       RichString_append(out, CRT_colors[CPU_NICE], buffer);
51       sprintf(buffer, "%5.1f%% ", this->values[3]);
52       RichString_append(out, CRT_colors[METER_TEXT], "hi:");
53       RichString_append(out, CRT_colors[CPU_IRQ], buffer);
54       sprintf(buffer, "%5.1f%% ", this->values[4]);
55       RichString_append(out, CRT_colors[METER_TEXT], "si:");
56       RichString_append(out, CRT_colors[CPU_SOFTIRQ], buffer);
57       sprintf(buffer, "%5.1f%% ", this->values[5]);
58       RichString_append(out, CRT_colors[METER_TEXT], "wa:");
59       RichString_append(out, CRT_colors[CPU_IOWAIT], buffer);
60       sprintf(buffer, "%5.1f%% ", this->values[6]);
61       RichString_append(out, CRT_colors[METER_TEXT], "st:");
62       RichString_append(out, CRT_colors[CPU_STEAL], buffer);
63       if (this->values[7]) {
64          sprintf(buffer, "%5.1f%% ", this->values[7]);
65          RichString_append(out, CRT_colors[METER_TEXT], "gu:");
66          RichString_append(out, CRT_colors[CPU_GUEST], buffer);
67       }
68    } else {
69       sprintf(buffer, "%5.1f%% ", this->values[2]);
70       RichString_append(out, CRT_colors[METER_TEXT], "sys:");
71       RichString_append(out, CRT_colors[CPU_KERNEL], buffer);
72       sprintf(buffer, "%5.1f%% ", this->values[0]);
73       RichString_append(out, CRT_colors[METER_TEXT], "low:");
74       RichString_append(out, CRT_colors[CPU_NICE], buffer);
75       if (this->values[3]) {
76          sprintf(buffer, "%5.1f%% ", this->values[3]);
77          RichString_append(out, CRT_colors[METER_TEXT], "vir:");
```

```
78            RichString_append(out, CRT_colors[CPU_GUEST], buffer);
79         }
80      }
81   }
```

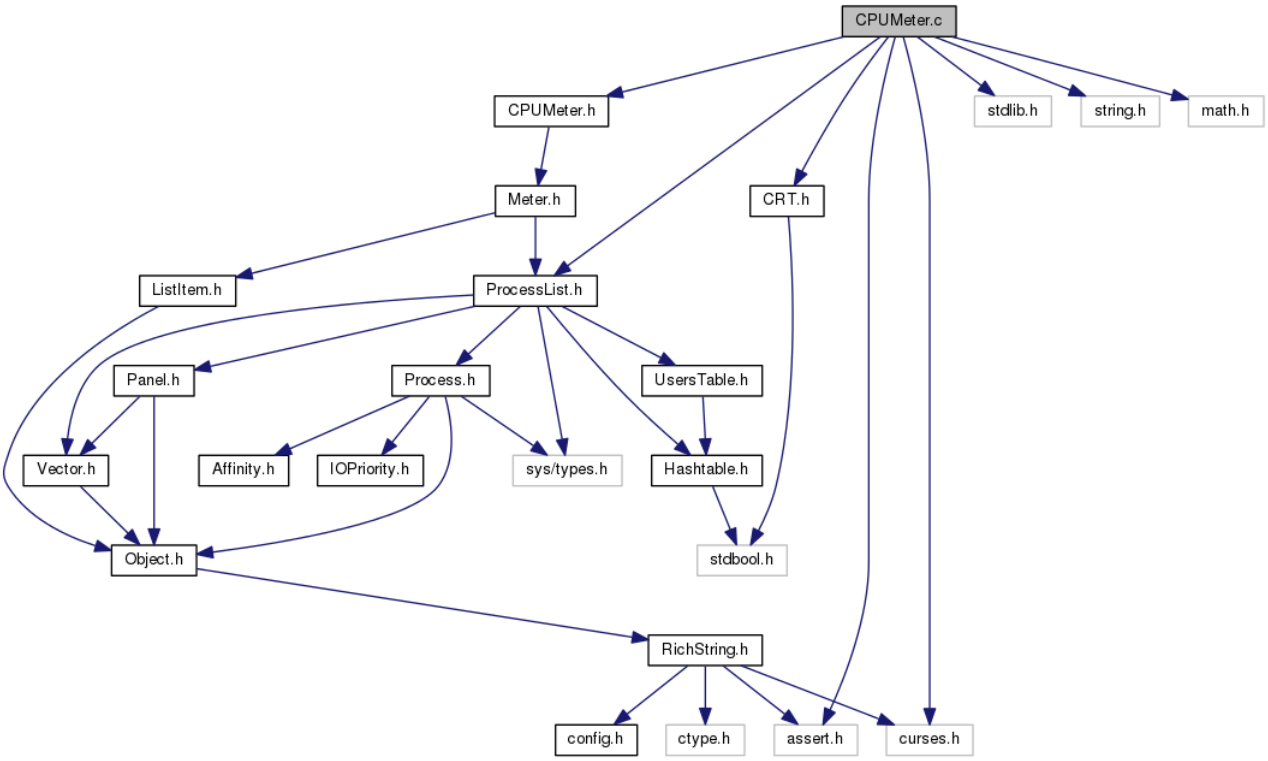

Рис. 6: Граф включения для файла CPUMeter.c

6 Имя устройства (хоста)

Этот модуль является самым простым. Его работа сводится фактически к одной строке, которая записывает имя хоста в переданный буфер. Функция gethostname является стандартной (POSIX), и она определена в заголовочном файле unistd.h. Вызов этой функции обёрнут другой функцией HostnameMeter_setValues, которая показана в листинге 5. Простота этой функции не отменяет её значимости, которая показана в графе включения на рисунке 7.

Листинг 6: Hostname Meter - имя устройства (хоста) (src/top/HostnameMeter.c)

```
1 static void HostnameMeter_setValues(Meter* this, char* buffer, int size) {
2    (void) this;
3    gethostname(buffer, size-1);
4 }
```

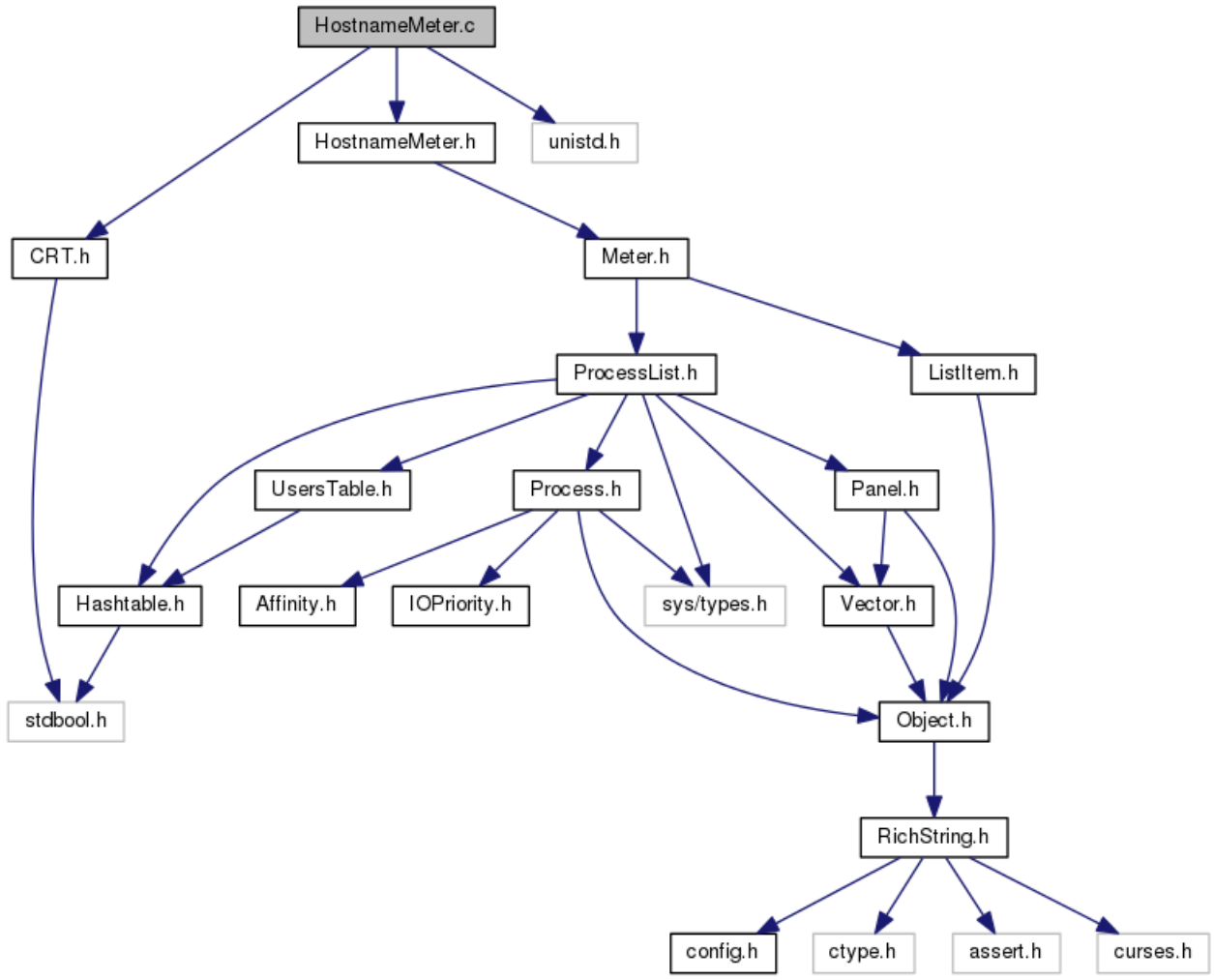

Рис. 7: Граф включения для файла HostnameMeter.c

7 Измерение средней загрузки

Большую часть задачи, как и раньше, берёт на себя ядро. В файловой системе proc оно создаёт файл loadavg. Этот файл состоит из пяти групп[2]:

- первое поле (число) — показывает использование процессора за последнюю минуту;
- второе поле (число) — показывает использование процессора за последние пять минут;
- третье поле (число) — показывает использование процессора за последние десять минут;
- четвёртое поле (число/число) — состоит из двух значений, разделённых слешем, первая часть значения поля показывает количество выполняющихся в данный момент процессов/потоков (это значение не может быть больше количества присутствующих в системе CPU), вторая часть поля отображает количество процессов присутствующих в системе;
- пятое поле (число) — хранит ID последнего запущенного в системе процесса (если для просмотра содержимого /proc/loadavg была использована команда cat, то будет значение PID именно программы cat).

В листинге 6 представлен отрывок файла LoadAverageMeter.c. Функция LoadAverageMeter_scan (стр. 1) как раз обеспечивает открытие файла loadavg (стр. 4) и чтение пяти рассмотренных выше полей (стр. 6), при этом четвёртое поле читается как два различных числа.

Функция LoadAverageMeter_setValues (стр. 14) обращается к LoadAverageMeter_scan и сохраняет полученные значения по загрузке процессора за последнюю минуту, пять минут и десять минут в буфер (стр. 16), переданный в качестве аргумента.

Для отображения значений используется функция LoadAverageMeter_display (стр 19). Она обеспечивает форматированный и цветной вывод для отображения загрузки процессора за последнюю минуту (стр. 22), пять минут (стр. 24) и десять минут (стр. 26).

Функции LoadMeter_setValues (стр. 30) и LoadMeter_display (стр 39) также обращаются к функции LoadAverageMeter_scan, но они отображают только загрузку за последнюю минуту, для просмотра изменения загрузки в режиме (относительно) реального времени.

Листинг 7: Load Average Meter - измерение средней загрузки (src/top/LoadAverageMeter.c)

```
1  static inline void LoadAverageMeter_scan(double* one, double* five, double*
       fifteen) {
2     int activeProcs, totalProcs, lastProc;
3     *one = 0; *five = 0; *fifteen = 0;
4     FILE *fd = fopen(PROCDIR "/loadavg", "r");
5     if (fd) {
6        int total = fscanf(fd, "%32lf %32lf %32lf %32d/%32d %32d", one, five,
             fifteen,
7           &activeProcs, &totalProcs, &lastProc);
8        (void) total;
9        assert(total == 6);
10       fclose(fd);
11    }
12 }
13
14 static void LoadAverageMeter_setValues(Meter* this, char* buffer, int size)
       {
15    LoadAverageMeter_scan(&this->values[2], &this->values[1], &this->values
          [0]);
16    snprintf(buffer, size, "%.2f/%.2f/%.2f", this->values[2], this->values
          [1], this->values[0]);
17 }
18
19 static void LoadAverageMeter_display(Object* cast, RichString* out) {
20    Meter* this = (Meter*)cast;
21    char buffer[20];
22    sprintf(buffer, "%.2f ", this->values[2]);
23    RichString_write(out, CRT_colors[LOAD_AVERAGE_FIFTEEN], buffer);
24    sprintf(buffer, "%.2f ", this->values[1]);
25    RichString_append(out, CRT_colors[LOAD_AVERAGE_FIVE], buffer);
26    sprintf(buffer, "%.2f ", this->values[0]);
27    RichString_append(out, CRT_colors[LOAD_AVERAGE_ONE], buffer);
28 }
29
30 static void LoadMeter_setValues(Meter* this, char* buffer, int size) {
31    double five, fifteen;
32    LoadAverageMeter_scan(&this->values[0], &five, &fifteen);
33    if (this->values[0] > this->total) {
34       this->total = this->values[0];
35    }
36    snprintf(buffer, size, "%.2f", this->values[0]);
37 }
38
39 static void LoadMeter_display(Object* cast, RichString* out) {
```

```
40    Meter* this = (Meter*)cast;
41    char buffer[20];
42    sprintf(buffer, "%.2f ", ((Meter*)this)->values[0]);
43    RichString_write(out, CRT_colors[LOAD], buffer);
44 }
```

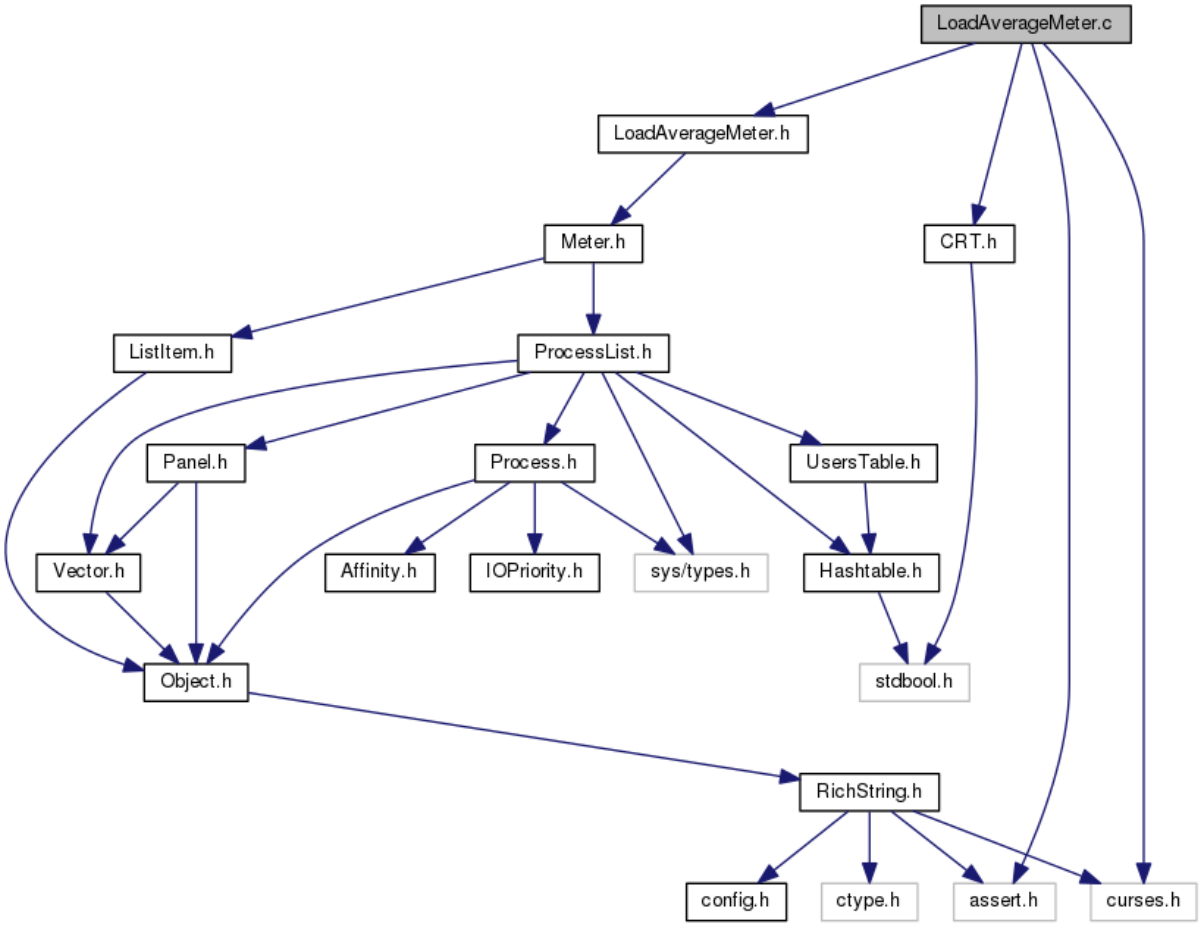

Рис. 8: Граф включения для файла LoadAverageMeter.c

8 Измерение уровня использования памяти

Сбор различных характеристик системы уже был показан в листинге 2. В листинге 8 представлены две функции.

Функция MemoryMeter_setValues (стр. 1) выводит в буфер следующую информацию:

- usedMem (стр. 2) - реально использующая в данный момент и зарезервированная системой память;
- buffersMem (стр. 3) - буферы в памяти это страницы памяти, зарезервированные системой для выделения их процессам, когда они затребуют этого, так же известна как heap-memory;
- cachedMem (стр. 4) - файлы, которые недавно были использованы системой/процессами и хранящиеся в памяти на случай если вскоре они снова потребуются.

Функция MemoryMeter_display (стр. 13) занимается оформленным выводом собранной информации. Граф включения представлен на рисунке 9.

Листинг 8: Memory Meter - измерение уровня использования памяти (src/top/MemoryMeter.c)

```
1  static void MemoryMeter_setValues(Meter* this, char* buffer, int size) {
2     long int usedMem = this->pl->usedMem;
3     long int buffersMem = this->pl->buffersMem;
4     long int cachedMem = this->pl->cachedMem;
5     usedMem -= buffersMem + cachedMem;
6     this->total = this->pl->totalMem;
7     this->values[0] = usedMem;
8     this->values[1] = buffersMem;
9     this->values[2] = cachedMem;
10    snprintf(buffer, size, "%ld/%ldMB", (long int) usedMem / 1024, (long int)
          this->total / 1024);
11 }
12
13 static void MemoryMeter_display(Object* cast, RichString* out) {
14    char buffer[50];
15    Meter* this = (Meter*)cast;
16    int k = 1024; const char* format = "%ldM ";
17    long int totalMem = this->total / k;
18    long int usedMem = this->values[0] / k;
19    long int buffersMem = this->values[1] / k;
20    long int cachedMem = this->values[2] / k;
21    RichString_write(out, CRT_colors[METER_TEXT], ":");
22    sprintf(buffer, format, totalMem);
```

```
23     RichString_append(out, CRT_colors[METER_VALUE], buffer);
24     sprintf(buffer, format, usedMem);
25     RichString_append(out, CRT_colors[METER_TEXT], "used:");
26     RichString_append(out, CRT_colors[MEMORY_USED], buffer);
27     sprintf(buffer, format, buffersMem);
28     RichString_append(out, CRT_colors[METER_TEXT], "buffers:");
29     RichString_append(out, CRT_colors[MEMORY_BUFFERS], buffer);
30     sprintf(buffer, format, cachedMem);
31     RichString_append(out, CRT_colors[METER_TEXT], "cache:");
32     RichString_append(out, CRT_colors[MEMORY_CACHE], buffer);
33 }
```

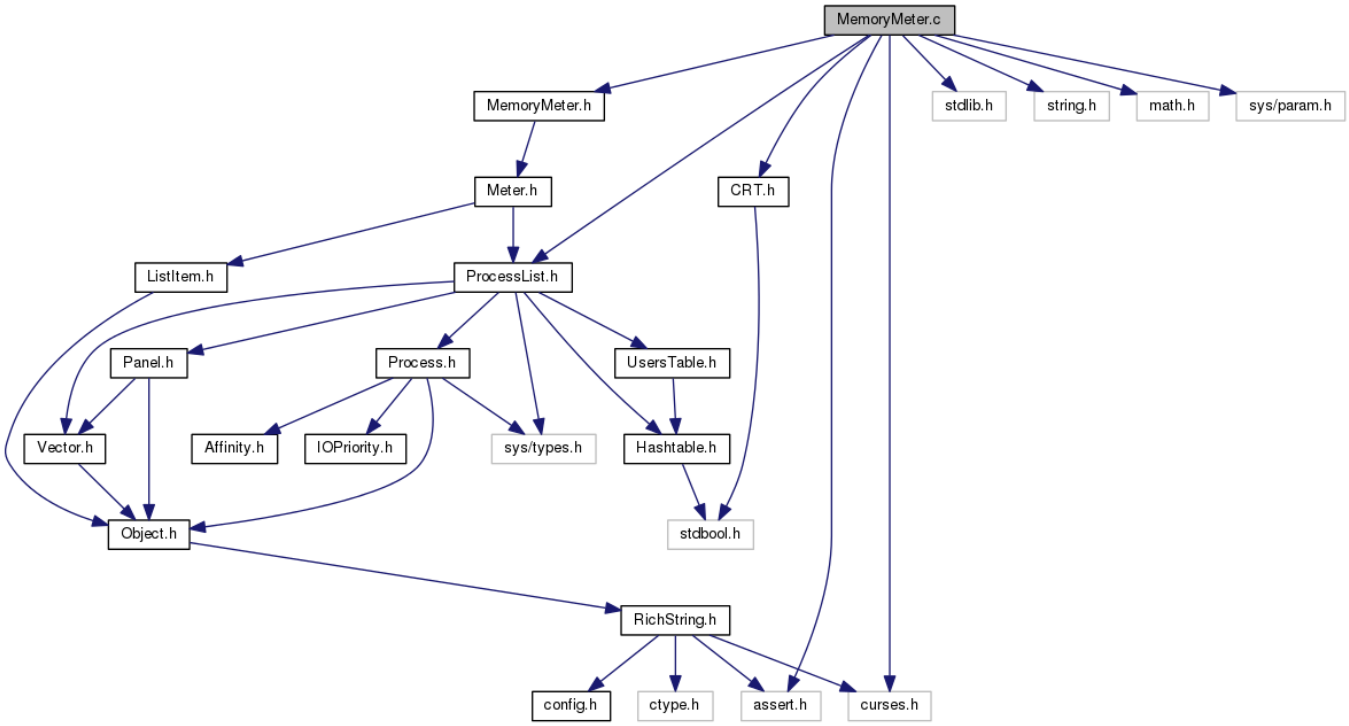

Рис. 9: Граф включения для файла MemoryMeter.c

9 Измерение уровня использования области подкачки

Если система приступает к запуску программу, которая требует больше оперативной памяти, чем доступно, то для решения этой задачи, используется технология swapping (”подкачка”). Суть этой технологии заключается в том, что некоторый объем данных (который не “помещается” в оперативную память) временно хранится на жестком диске, в то время как другая часть данных обрабатывается.

В Linux оперативная память делится на разделы, называемые страницами. Swapping (подкачка) – это процесс во время которого страницы памяти копируются на специально сконфигурированный для этого раздел диска, называемый swap space (раздел подкачки, может быть как и файлом, так и разделом жесткого диска), для освобождения ОЗУ. Совокупные размеры физической памяти и раздела подкачки – это объем имеющийся виртуальной памяти.

В листинге 2 мы рассматривали процедуру сбора системной информации, в листинге 9 нас интересуют три функции работы со свапом.

В функции SwapMeter_humanNumber (стр. 2) размер раздела подкачки приводится к виду, легко воспринимаемому человеком. Если объём превышает 10 гигабайт, то число отображается как соответствующее количество гигабайт (стр 4). Если число превышает 10 мегабайт, то оно отображается как соответствующее количество мегабайт (стр 6). Если ни одно из предыдущих правил не сработало, то объём отображается в килобайтах (стр. 8).

Функция SwapMeter_setValues (стр. 11) выводит информацию о подкачке в буфер, переданный в качестве параметра. Для этого используется общий объём свапа (стр. 13) и занятый объём (стр. 12).

Функция SwapMeter_display (стр 18) обеспечивает форматированный вывод всей собранной информации. Граф включения представлен на рисунке 10.

Листинг 9: Swap Meter - измерение уровня использования области подкачки (src/top/SwapMeter.c)

```
1  /* NOTE: Value is in kilobytes */
2  static void SwapMeter_humanNumber(char* buffer, const long int* value) {
3     if (*value >= 10*GIGABYTE)
4        sprintf(buffer, "%ldG ", *value / GIGABYTE);
5     else if (*value >= 10*MEGABYTE)
6        sprintf(buffer, "%ldM ", *value / MEGABYTE);
7     else
8        sprintf(buffer, "%ldK ", *value);
9  }
10
```

```
11 static void SwapMeter_setValues(Meter* this, char* buffer, int len) {
12    long int usedSwap = this->pl->usedSwap;
13    this->total = this->pl->totalSwap;
14    this->values[0] = usedSwap;
15    snprintf(buffer, len, "%ld/%ldMB", (long int) usedSwap / MEGABYTE, (long
          int) this->total / MEGABYTE);
16 }
17
18 static void SwapMeter_display(Object* cast, RichString* out) {
19    char buffer[50];
20    Meter* this = (Meter*)cast;
21    long int swap = (long int) this->values[0];
22    long int total = (long int) this->total;
23    RichString_write(out, CRT_colors[METER_TEXT], ":");
24    SwapMeter_humanNumber(buffer, &total);
25    RichString_append(out, CRT_colors[METER_VALUE], buffer);
26    SwapMeter_humanNumber(buffer, &swap);
27    RichString_append(out, CRT_colors[METER_TEXT], "used:");
28    RichString_append(out, CRT_colors[METER_VALUE], buffer);
29 }
```

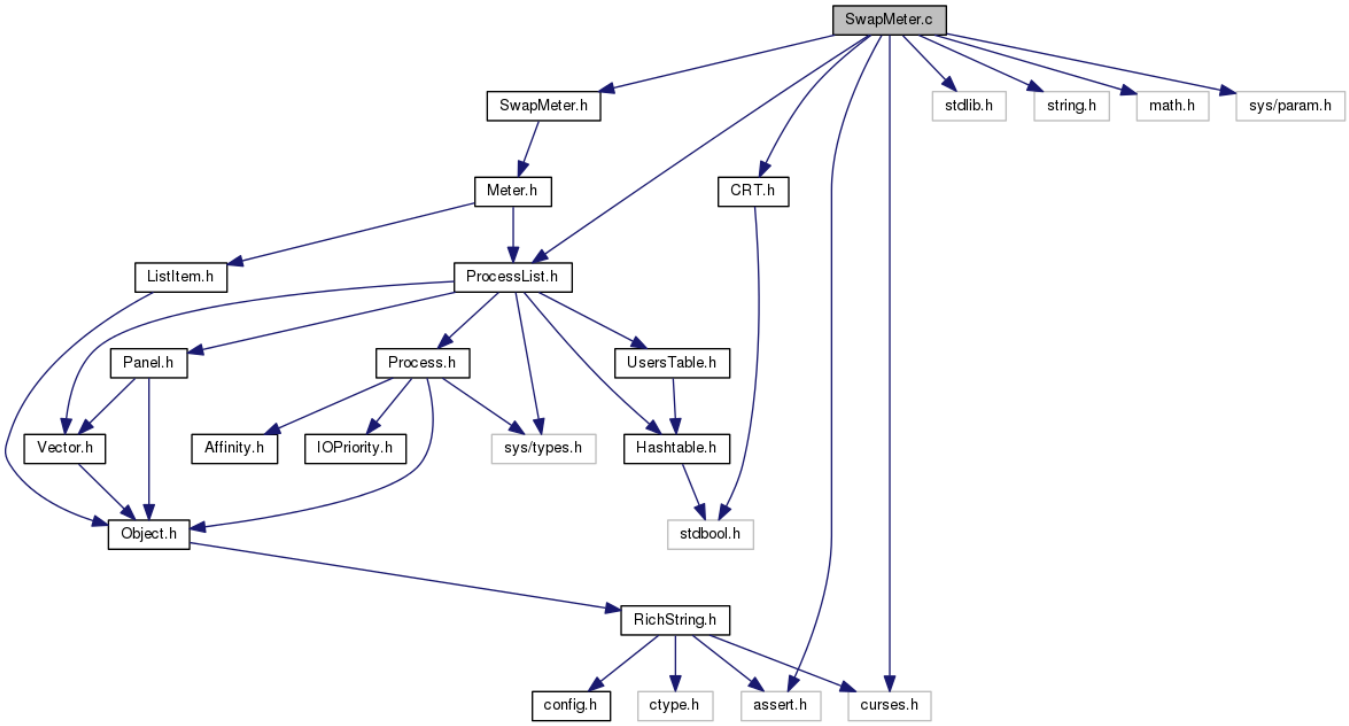

Рис. 10: Граф включения для файла SwapMeter.c

10 Мониторинг процессов

Как мы уже говорили, все процессы представлены своей директорией в файловой системе proc. вывод этой информации можно изучить в листинге 10.

Функция TasksMeter_setValues (стр. 1) занимается в выводом в буфер (стр. 5) информации о процессах. Этой информацией является общее количество процессов (стр. 3) и количество запущенных в данный момент процессов (стр. 4). Как мы говорили выше, количество запущенных процессов не может превышать количество процессоров.

Функция TasksMeter_display (стр. 8) обеспечивает форматированный вывод информации о процессах. При этом стоит обратить внимание, что она выводит не только процессы пользователя (стр. 22) но и процессы ядра (стр. 28), которые могут заниматься, к примеру, управлением кэшами.

Граф включений представлен на рисунке 11.

Листинг 10: Tasks Meter - мониторинг процессов (src/top/TasksMeter.c)

```
1  static void TasksMeter_setValues(Meter* this, char* buffer, int len) {
2     ProcessList* pl = this->pl;
3     this->total = pl->totalTasks;
4     this->values[0] = pl->runningTasks;
5     snprintf(buffer, len, "%d/%d", (int) this->values[0], (int) this->total);
6  }
7
8  static void TasksMeter_display(Object* cast, RichString* out) {
9     Meter* this = (Meter*)cast;
10    ProcessList* pl = this->pl;
11    char buffer[20];
12    sprintf(buffer, "%d", (int)(this->total - pl->userlandThreads - pl->
         kernelThreads));
13    RichString_write(out, CRT_colors[METER_VALUE], buffer);
14    int threadValueColor = CRT_colors[METER_VALUE];
15    int threadCaptionColor = CRT_colors[METER_TEXT];
16    if (pl->highlightThreads) {
17       threadValueColor = CRT_colors[PROCESS_THREAD_BASENAME];
18       threadCaptionColor = CRT_colors[PROCESS_THREAD];
19    }
20    if (!pl->hideUserlandThreads) {
21       RichString_append(out, CRT_colors[METER_TEXT], ", ");
22       sprintf(buffer, "%d", (int)pl->userlandThreads);
23       RichString_append(out, threadValueColor, buffer);
24       RichString_append(out, threadCaptionColor, " thr");
25    }
```

```
26    if (!pl->hideKernelThreads) {
27       RichString_append(out, CRT_colors[METER_TEXT], ", ");
28       sprintf(buffer, "%d", (int)pl->kernelThreads);
29       RichString_append(out, threadValueColor, buffer);
30       RichString_append(out, threadCaptionColor, " kthr");
31    }
32    RichString_append(out, CRT_colors[METER_TEXT], "; ");
33    sprintf(buffer, "%d", (int)this->values[0]);
34    RichString_append(out, CRT_colors[TASKS_RUNNING], buffer);
35    RichString_append(out, CRT_colors[METER_TEXT], " running");
36 }
```

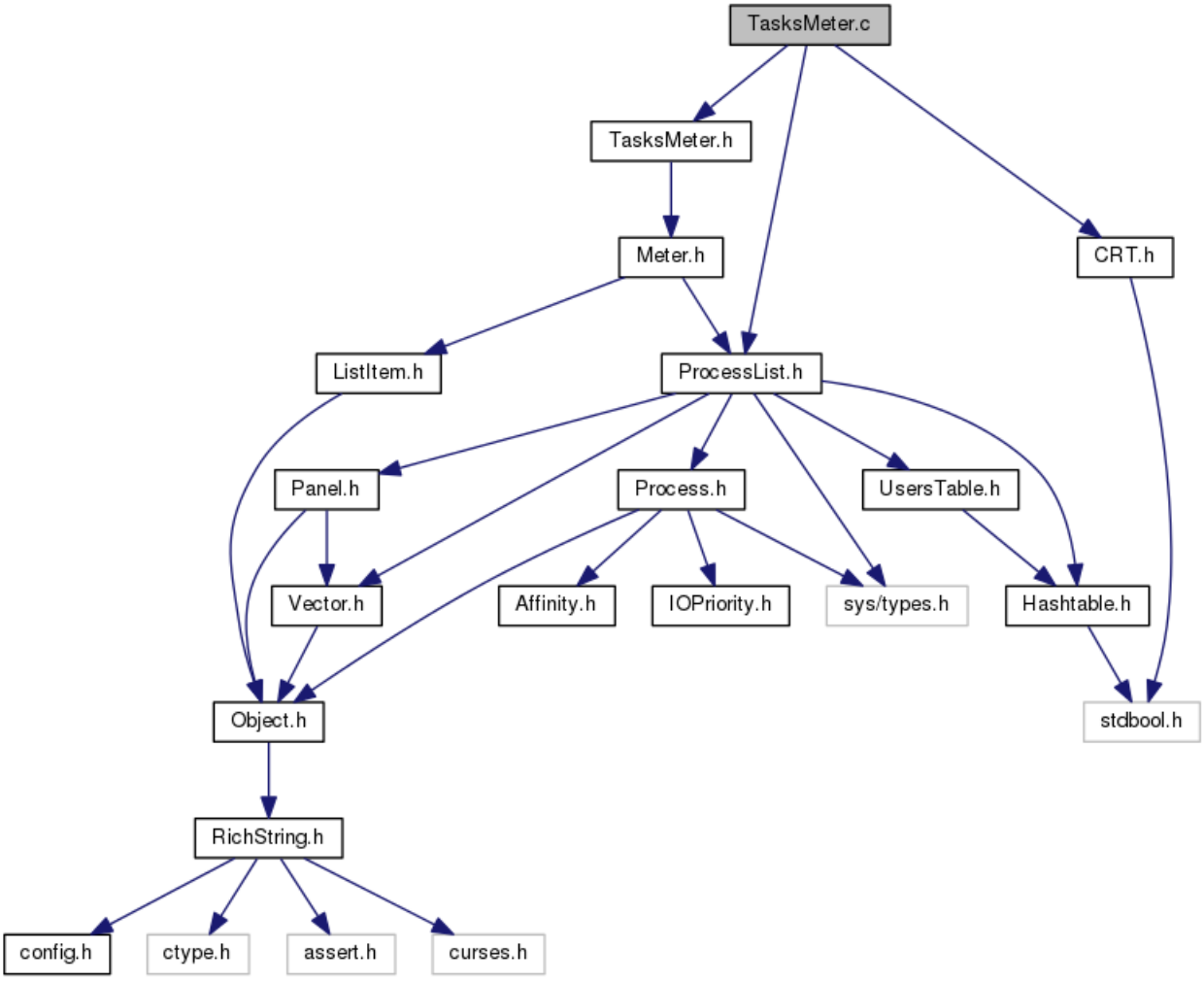

Рис. 11: Граф включения для файла TasksMeter.c

11 Измерение времени работы системы

Время работы системы также хранится в файловой системе proc в файле uptime. В листинге 11 представлена функция UptimeMeter_setValues (стр. 1), которая открывает этот файл (стр. 3) и считывает оттуда значение (стр. 5). Стоит отметить, что на самом деле в этом файле хранится два числа.

- первое показывает количество секунд, прошедших с момента включения компьютера (это значение используется в дальнейшем)
- второе показывает количество времени (тоже в секундах), которое система провела в бездействии (в т.ч. ожидая завершения операций ввода-вывода); на многоядерных системах это число складывается из времени бездействия каждого ядра, так что второе число по значению может обогнать первое.

Число секунд с момента включения приводится к виду, удобному для человеческого восприятия - в буфер выводится (стр. 27) количество дней, часов, минут и секунд работы.

Листинг 11: Uptime Meter - измерение времени работы системы (src/top/UptimeMeter.c)

```
1  static void UptimeMeter_setValues(Meter* this, char* buffer, int len) {
2     double uptime = 0;
3     FILE* fd = fopen(PROCDIR "/uptime", "r");
4     if (fd) {
5        fscanf(fd, "%64lf", &uptime);
6        fclose(fd);
7     }
8     int totalseconds = (int) ceil(uptime);
9     int seconds = totalseconds % 60;
10    int minutes = (totalseconds/60) % 60;
11    int hours = (totalseconds/3600) % 24;
12    int days = (totalseconds/86400);
13    this->values[0] = days;
14    if (days > this->total) {
15       this->total = days;
16    }
17    char daysbuf[15];
18    if (days > 100) {
19       sprintf(daysbuf, "%d days(!), ", days);
20    } else if (days > 1) {
21       sprintf(daysbuf, "%d days, ", days);
22    } else if (days == 1) {
23       sprintf(daysbuf, "1 day, ");
24    } else {
25       daysbuf[0] = '\0';
```

```
26     }
27     snprintf(buffer, len, "%s%02d:%02d:%02d", daysbuf, hours, minutes,
           seconds);
28 }
```

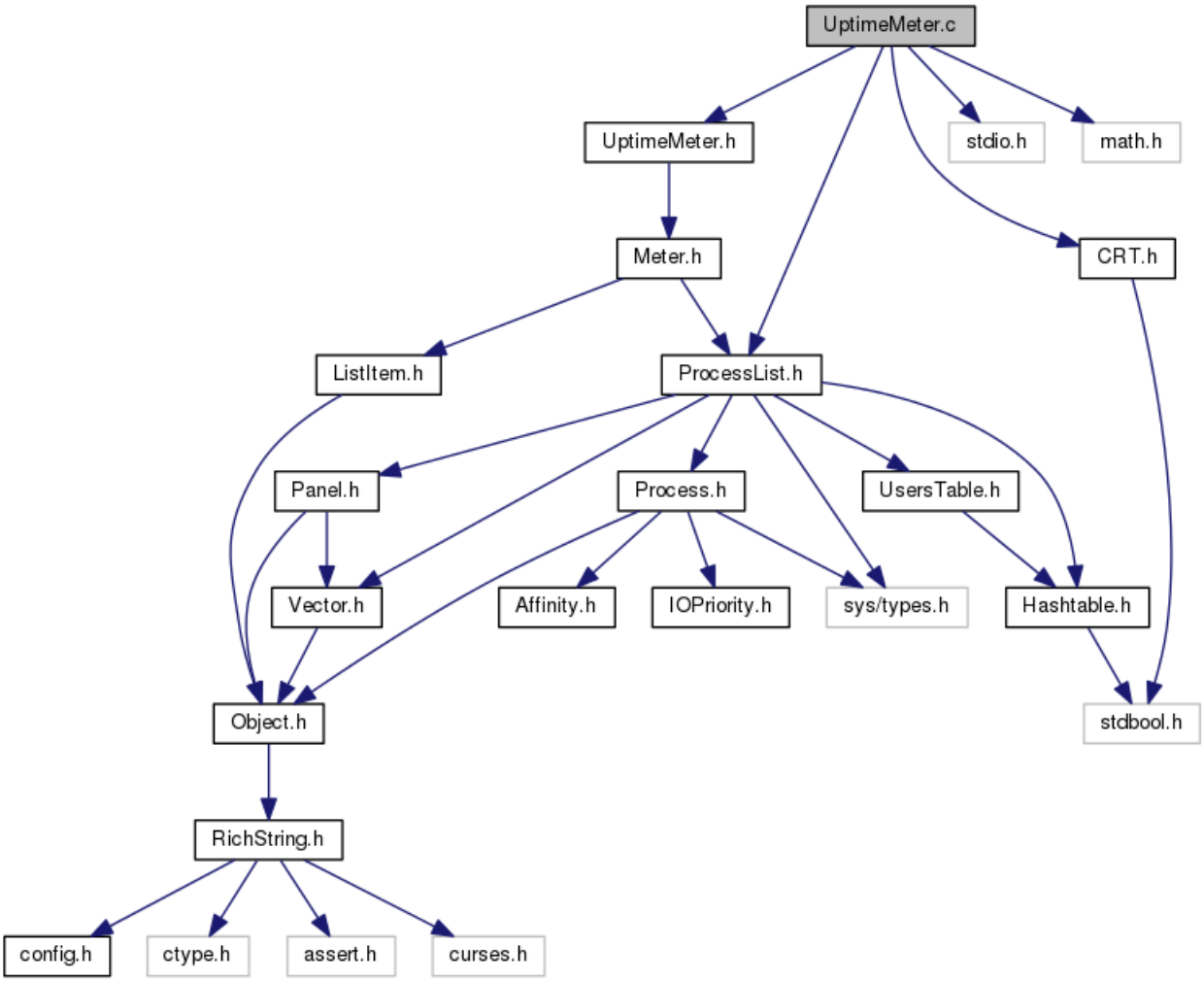

Рис. 12: Граф включения для файла UptimeMeter.c

12 Модификация для работы с процессом

Как уже говорилось во введении, на верхнем уровне процессы представляют собой директории, именованные в соответствии с их pid. Рассмотрим, какую информацию можно получить из файловой системы proc, а потом добавим рассматриваемой утилите возможность работать с процессом, переданным в качестве параметра по его pid.

Некоторые файлы и директории из ProcFS[2]:

- /proc/PID/cmdline – аргументы командной строки (где PID – идентификатор процесса или self);
- /proc/PID/environ – переменные окружения для данного процесса;
- /proc/PID/status – статус процесса;
- /proc/PID/fd – директория, содержащая символьные ссылки на каждый открытый файловый дескриптор;
- /proc/cpuinfo – информация о процессоре (производитель, модель, поколение и т.п.);
- /proc/cmdline – параметры, передаваемые ядру при загрузке;
- /proc/uptime – количество секунд, прошедших с момента загрузки ядра и проведенных в режиме бездействия;
- /proc/version – содержит информацию о версии ядра, компилятора и другую информацию, связанную с загруженным ядром.

Приведём несколько примеров использования procfs.

Следующая команда демонстрирует, как из procfs можно получить информацию о текущей рабочей директории процесса (3165 — номер pid'а процесса)

```
# ls -la /proc/3165/cwd
lrwxrwxrwx 1 clamav clamav 0 Авг 18 16:07 /proc/3165/cwd -> /var/lib/clamav
```

Далее, можно вывести все переменные процесса

```
# cat /proc/3165/environ | strings
ReceiveTimeout=30
CONSOLE=/dev/console
SELINUX_INIT=YES
TERM=linux
rootmnt=/root
PidFile=/var/run/clamav/freshclam.pid
```

```
NotifyClamd=/etc/clamav/clamd.conf
LogTime=no
INIT_VERSION=sysvinit-2.86
init=/sbin/init
DNSDatabaseInfo=current.cvd.clamav.net
AllowSupplementaryGroups=false
PATH=/sbin:/usr/sbin:/bin:/usr/bin
LogSyslog=false
DatabaseMirror=database.clamav.net db.local.clamav.net
runlevel=2
RUNLEVEL=2
PWD=/
VERBOSE=no
DatabaseOwner=clamav
CompressLocalDatabase=no
previous=N
PREVLEVEL=N
LogVerbose=false
MaxAttempts=5
ScriptedUpdates=yes
Foreground=false
Checks=24
SHLVL=3
HOME=/
DatabaseDirectory=/var/lib/clamav/
LogFacility=LOG_LOCAL6
UpdateLogFile=/var/log/clamav/freshclam.log
LogFileMaxSize=0
ConnectTimeout=30
Debug=false
_=/sbin/start-stop-daemon
```

Теперь можно получить статистику процесса

```
$ cat /proc/1742/status
Name:   bash
State:  S (sleeping)
Tgid:   3515
```

```
Pid:    3515
PPid:   3452
TracerPid:      0
Uid:    1000    1000    1000    1000
Gid:    100     100     100     100
FDSize: 256
Groups: 16 33 100
VmPeak:     9136 kB
VmSize:     7896 kB
VmLck:         0 kB
VmHWM:      7572 kB
VmRSS:      6316 kB
VmData:     5224 kB
VmStk:        88 kB
VmExe:       572 kB
VmLib:      1708 kB
VmPTE:        20 kB
Threads:        1
SigQ:   0/3067
SigPnd: 0000000000000000
ShdPnd: 0000000000000000
SigBlk: 0000000000010000
SigIgn: 0000000000384004
SigCgt: 000000004b813efb
CapInh: 0000000000000000
CapPrm: 0000000000000000
CapEff: 0000000000000000
CapBnd: ffffffffffffffff
Cpus_allowed:   00000001
Cpus_allowed_list:      0
Mems_allowed:   1
Mems_allowed_list:      0
voluntary_ctxt_switches:        150
nonvoluntary_ctxt_switches:     545
```

Самое полезное в данном выводе, это поле Threads. Оно показывает количество потоков у процесса (ещё эту информацию можно получить через утилиту ps)[3]. Теперь остаётся внести изменения в исследуемую утилиту.

Поле с потоками называется NLWP (Number of Light-Weight Processes). Это название пошло со времён ОС Солярис.

Для работы этого функционала в структуре ProcessList_ (объявлена в заголовочном файле Process.h) объявлена переменная nlwp типа long int (см. стр. 33 в листинге 1). Чтение из status-файла происходит в функции ProcessList_readStatFile, листинг которой представлен в листинге 12.

Листинг 12: Чтение stat-файла процесса (src/top/ProcessList.c)

```
1  static bool ProcessList_readStatFile(Process *process, const char* dirname,
       const char* name, char* command) {
2     char filename[MAX_NAME+1];
3     snprintf(filename, MAX_NAME, "%s/%s/stat", dirname, name);
4     FILE* file = fopen(filename, "r");
5     if (!file)
6        return false;
7
8     static char buf[MAX_READ];
9
10    int size = fread(buf, 1, MAX_READ, file);
11    if (!size) { fclose(file); return false; }
12
13    //assert(process->pid == atoi(buf));
14    char *location = strchr(buf, ' ');
15    if (!location) { fclose(file); return false; }
16
17    location += 2;
18    char *end = strrchr(location, ')');
19    if (!end) { fclose(file); return false; }
20
21    int commsize = end - location;
22    memcpy(command, location, commsize);
23    command[commsize] = '\0';
24    location = end + 2;
25
26    int num = sscanf(location,
27       "%c %d %u %u %u "
28       "%d %lu "
29       "%*u %*u %*u %*u "
30       "%llu %llu %llu %llu "
31       "%ld %ld %ld "
32       "%*d %*u %*u %*d %*u %*u %*u %*u %*u %*u %*u %*u %*u %*u %*u %*u %*u "
33       "%d %d",
34       &process->state, &process->ppid, &process->pgrp, &process->session, &
           process->tty_nr,
```

```
35        &process->tpgid, &process->flags,
36        &process->utime, &process->stime, &process->cutime, &process->cstime,
37        &process->priority, &process->nice, &process->nlwp,
38        &process->exit_signal, &process->processor);
39    fclose(file);
40    return (num == 16);
41 }
```

Функция получает путь к proc (dirname) и номер процесса (name) в качестве параметров, из которых складывается полный путь к stat-файлу (стр 3). Этот файл открывается на чтение (стр стр. 4) и после некоторых проверок из него выбираются значения, в т.ч. количество процессов (стр. 37).

Запуск модифицированной версии, для отслеживания потоков firefox

```
sam@spb:~/tmp/top$ ./top --pid 19771
```

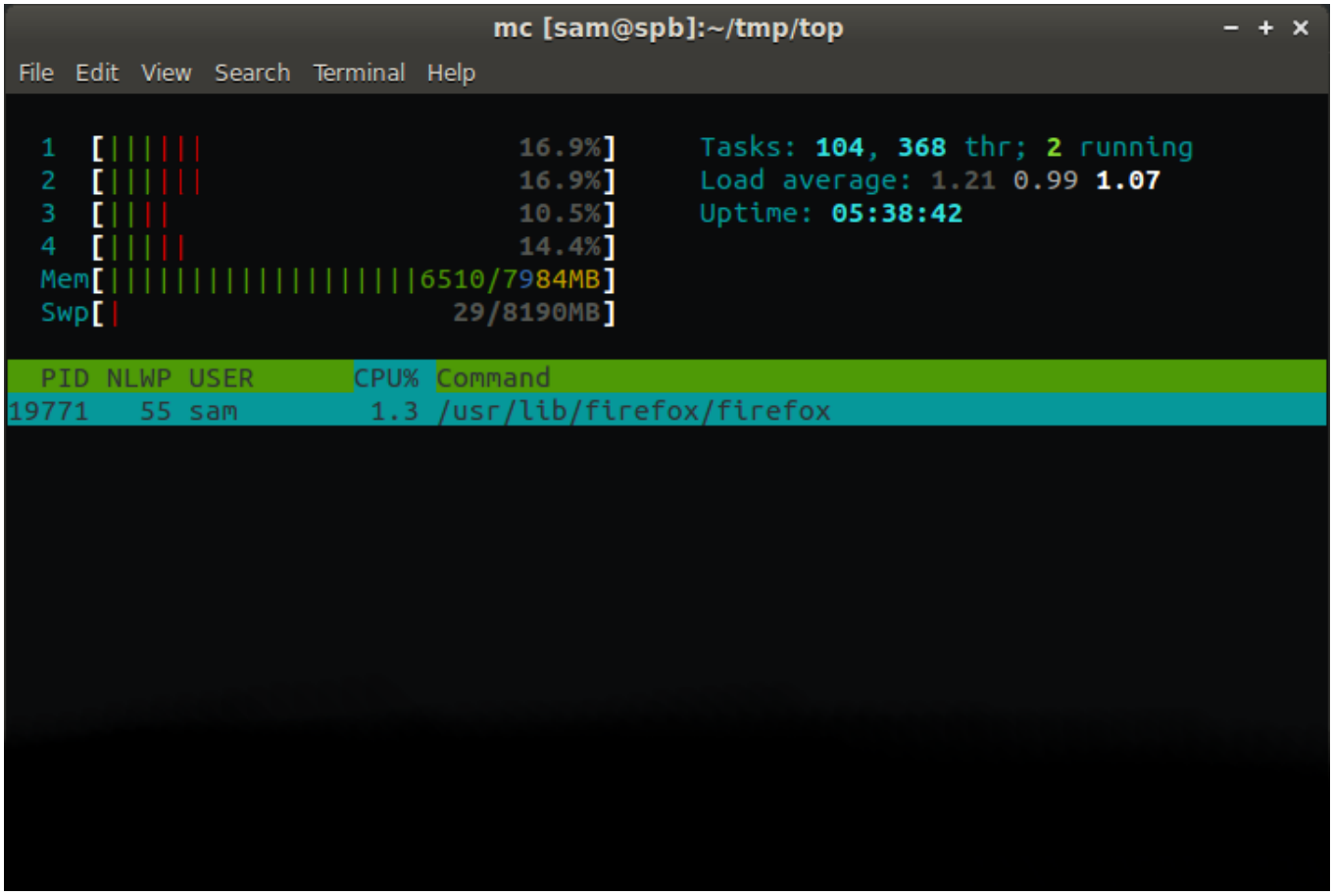

Рис. 13: Количество потоков у процесса с pid 19771

Заключение

В данной работе нами была изучена программа htop, являющееся расширенной версией стандартной утилиты top. Мы выпустили из рассмотрения особенности, связанные с графическим выводом (на базе библиотеки ncurces), сосредоточившись на основных функциональных возможностях.

Изучив исходный код, мы убедились, что никакие специальные системные вызовы утилита не использует, только самые широко распространённы, вроде чтения из файла и вывод на экран (но они используются так часто, что их систематизация для данного отчёта оказалась крайне затруднительной).

Основным выводом является тот факт, что в отличии от Windows, Linux предоставляет удобный механизм сбора системной информации через файловую систему proc, который активно используется утилитами вроде ps, htop (стр. 5 листинг 2; стр. 18 листинг 3; стр. 136 листинг 3; стр. 201 листинг 3; стр. 4 листинг 7; стр. 3 листинг 11; стр. 26 листинг 13).

Файловая система proc обладает обширными возможностями по конфигурированию linux, в то же время её использование требует предельной осторожности, так как попытка записи в некоторые файлы может повредить файловую систему или привести к краху системы.

Список литературы

1. Яремчук С. А. Linux Mint на 100 %. – СПб.: Питер, 2011. – 240 е.: ил. — (Серия «На 100 %»). ISBN: 978-5-49807-803-8.
2. HowTo: Troubleshoot with linux 'top' command. David Van Rood (dowdandassociates.com).
3. Арнольд Роббинс. Linux. Программирование в примерах – СПб.: КУДИЦ-Пресс, 2006 – 256 стр.