摘 要

为用户提供一款能够便捷访问系统资源的轻量级系统资源管理器，通过/proc文件系统、linux 内核 api 等获取数据，存储并显示，可通过 web 端获取数据，可运行在本地主机，并通过公网 web 端访问获取数据。

关键词：系统监控 内核 API WEB 端

目 录

说明:

1、报告要求 7000 字以上。

2、本模板仅为基本参考，请各位同学根据个人情况进行目录结构扩展。

3、报告正文必须双面打印。

第一章 需求分析与实施方案可行性研究

1.1 需求分析与建模

在 Linux 系统下，监控显示每一个进程的 CPU 使用量、内存使用量、交换内存、缓存大小、缓冲区大小、流程 PID、用户、命令等信息，用程序实现针对上面指标的显示列表、计费，用于控制特别进程的使用，为云计算调度，提供支持。

整个 LINUX 系统：

- 物理内存总量
- 使用的物理内存总量
- 空闲内存总量
- 用作内核缓存的内存量
- 交换区总量
- 使用的交换区总量
- 空闲交换区总量
- 缓冲的交换区总量
- 每一个进程的 CPU 使用量：
- 上次更新到现在的 CPU 时间占用百分比
- 进程使用的 CPU 时间总计，单位秒
- 每一个进程的内存使用量：
- 进程使用的物理内存百分比
- 进程使用的、未被换出的物理内存大小，单位 kb。
- 进程使用的虚拟内存总量，单位 kb
- 可执行代码占用的物理内存大小，单位 kb
- 可执行代码以外的部分(数据段+栈)占用的物理内存大小，单位 kb
- 每一个进程的交换内存：
- 进程使用的虚拟内存中，被换出的大小，单位 kb
- 每个进程的流程 PID：
- 进程 ID

- 父进程 ID
- 每个进程的用户：
- Real user name
- 进程所有者的用户 id
- 进程所有者的用户名
- 进程所有者的组名
- 每个进程的命令：
- 命令名/命令行

为了方便用户在无显示的主机上使用，有必要使用基于终端的显示方式。

为了方便用户方便的查看系统资源，可以增加 web 端访问方式。

为了方便用户能够在无公网 ip 的主机上通过 web 端访问，可视情况加入内网穿透功能。

1.2复杂工程问题归纳

1.2.1 如何获取到所需信息

本项目所需的信息，诸如 cpu 使用情况、内存剩余量等数据，基本都由内核管理，如何直接或间接的获取到所需信息是本项目的核心。

1.2.2 如何处理、组织并显示获取的信息

在得到数据后，如何对数据进行处理，把部分间接信息转化为所需信息，以及对处理后的信息进行组织，以方便计费，这是本项目的重要任务，组织后的信息如何展示给用户，以便用户及时了解系统资源使用情况，也是本项目的重要任务。

1.2.3 如何通过 web 端实时刷新获取信息

当用户无法通过本地终端访问主机时，如何在远程便捷的了解到系统资源使用情况，无疑 web 是最方便的方式，支持多类型终端设备，也无需额外辅助程序即可访问，那么如何通过 web 将本地主机资源情况展示给用户则是本项目的一个扩展性难题。

1.2.4 如何为云计算调度提供支持

本项目的目的是针对特殊进程通过获取的信息进行计费，用于控制特殊进程的使用，为云计算调度提供支持，故而只是普通的获取数据并实时显示当前数据明显是不够的。

1.3针对复杂工程问题的方案实现

1.3.1 如何获取到所需信息

要获取到诸如 cpu 内存使用情况之类的数据，不通过内核获取是不现实的，经过查阅资料，并对 Linux 平台的主流系统资源管理器的源码进行分析后，我们决定通过 Linux 提供的/proc 文件系统来获取系统资源，proc 文件系统是一个伪文件系统，它只存在内存当中，而不占用外存空间。它以文件系统的方式为访问系统内核数据的操作提供接口。用户和应用程序可以通过 proc 得到系统的信息，并可以改变内核的某些参数。由于系统的信息，如进程，是动态改变的，所以用户或应用程序读取 proc 文件时，proc 文件系统是动态从系统内核读出所需信息并提交的。

从时间上分析，proc 文件系统的信息刷新速度为 1 个 jiffies，视硬件平台的不同，1jeffies 约为 1/1000 到 1/100 秒，时间数量级上完全满足我们的需求。

从含有的信息分析，proc 文件系统的 stat 含有 cpu 情况和部分进程信息，sys 文件夹下则含有内核参数，<pid>文件夹下则含有进程的所有者、内存使用、cpu 占用、cmd 命令行、网络总收发、cache、交换分区使用、io 读写等信息，已经足够我们用于统计计费。

1.3.2 如何处理、组织并显示获取的信息

对于得到的数据，往往只有纯粹的数字，有些甚至连项名都没有，我们需要根据/proc 提供的数据，根据内核的配置映射到对应的名目，这个过程需要查阅大量的资料，且不同版本的内核由于参数配置不一样，部分数据还要根据内核版本单独处理，这个过程会极其费时，对于开发周期非常不利，所以这部分我们并没有重头开始造轮子，而是选择了调用 linux 内核提供的 API，通过/proc 文件系统和内核 API，将部分操作黑盒化。同时使之对用户透明，一方面这加速了开发周期，另一方面这在减小了程序体积的同时，也增强了系统数据的安全性和保密性。而对于非间接数据，如内存使用量等，在系统内均采用字节或其他单位表示，直接展示不便于用户直观查看，我们根据其大小将其采用 GB、MB、KB、B 等单位存储。

我们需要计算的是一个进程整个生命周期的的费用，故而有些数据需要处理为累加和或者计算出平均值。

对于数据的组织，为了便于存储和读取，我们将其根据数据类型直接写入文件。对于数据的显示，我们参考了主流的轻量级系统资源浏览器 top，选择了使

用终端来显示数据，这样就可以避免由于 GUI 组件引入，而带来的程序明显增大。而基于终端的显示，如果采用直接输出的话，会产生滚屏，显示效果极其恶劣，以至于无法让用户看清数据，所以这里我们采用的 linux 系统自带的 api termios 和一个第三方库 libncurses 来控制终端输入，以实现类似于 top 的显示效果。

1.3.3 如何通过 web 端实时刷新获取信息

访问 Web 端获取信息，要根据主机是否有公网 ip 来处理，如果主机具有公网 ip，能通过公网访问，只需在主机上运行任意一款主流的 web 服务器，通过访问域名或 ip 获取 html 文件后，通过 html 来定时获取服务器上格式化好的数据后直接展示，格式化好的数据可以加快 web 端的开发速度。而对于无公网 ip 的主机，由于无法直接通过公网 ip 访问，必须使用具有公网 ip 的服务器做反向代理，考虑到主机的配置便易，我们采用了 ngrok 来做反向代理，ngrok 采用 go 语言编写，跨平台容易，同时仅需一个配置文件即可完成配置，反向代理的过程增强了本地主机的安全性和本地主机数据的保密性。

1.3.4 如何为云计算调度提供支持

本项目为云计算调度提供支持的方式，是表现在对特殊进程的计费上，故而我们需要对特殊进程的整个生命周期的数据进行统计，而不是单单显示记录某一时刻的数值。

针对不同的指标，采用不同的处理。

对于 cpu 占用时间，以 jiffies 为单位进行计算，进程每运行 1 个 jiffies 则改数值+1；对于内存使用，整个进程的大小分为两部分，包含常驻内存部分和交换到磁盘的部分，常驻内存部分就是我们通常说到的占用内存大小，交换到磁盘的部分则是我们所说的占用交换区大小，无论是内存大小还是交换区大小，传统系统资源监视应用的标准计费方式都是直接读取/proc/<pid>/statm 的数值后记录显示，即都只记录莫值，这显然无法对云计算调度提供支持，故而我们选择通过记录进程整个生命周期的内存占用量和交换区占用量后，算出平均值，提供给云计算调度程序，以此减少误差；对于 IO 读写量等数据，则通过记录其总的 IO 读入量和写入量。

同时为了同时提供系统资源监视应用的功能和为云计算调度提供支持的功能，我们直接在系统资源监视应用的基础上，构建一个专门为云计算调度提供支持的模块，利用前者已获得的数据，通过二次加工，转换成能为云计算调度提供支持的数据，并利用这个模块进行单独存储和显示。

第二章 方案设计与实现

2.1 方案设计

总体上，程序采用 c 语言开发，一方面 c 语言便于调用 linux 内核的 api，另一方面 c 语言开发可以在尽可能减小程序体积的同时降低内存和 cpu 占用率，同时保证了程序的稳定性和运行速度。从/proc 文件系统获取所需信息，经程序处理后，在终端显示，并生成文件交给 web 服务器，然后由 ngrok 反向代理后可通过公网 web 端访问。

2.1.1 数据获取

数据的读取我们直接从/proc 文件系统读取原始数据，然后调用 linux 内核 api 将部分间接数据转换为所需数据。

/proc 文件系统由内核生成并维护，可保证数据的可靠性，内核动态更新/proc 的时间为 1 个 jiffies,，根据硬件平台的不同大约为 1/100 秒到 1/1000 秒，刷新时间数量级上已经足够，目前主流的 top 等系统资源监视器的刷新时间也仅为 1s~3s。

根据不同内核版本，/proc 提供了下述但不仅限于下述信息，用于查看进程的数据信息，查看与特定的内核子系统无关的一般信息，如/proc/iomem, /proc/ioports, /proc/interrupts，查看网络信息，/proc/net 子目录提供了内核的各种网络选项的有关数据，查看和修改系统控制参数，由/proc/sys 子目录提供。例，可以通过 cat /proc/sys/vm/swappiness 查看交换算法在换出页时的积极程度。

proc 文件系统，使得内核可以生成与系统状态和配置有关的信息。该信息可以由用户和系统程序从普通文件读取，而无需专门的工具与内核通信。我们可以采用处理 1 个普通文本的方式处理/proc 文件系统中的文件，可以加快开发周期。

2.1.2 数据存储

我们在设计途中首先想到的是将数据存储到数据库中，通过构建表来更规范的管理相关数据，数据库选用 MySql，MySql 是相比之下较为轻量级的数据库，且提供 C 语言 API，但方案设计完成后，我们却意识到作为一款系统资源监视器，我们考虑的不应该是如何存储数据，而是如何尽快的准确的将资源占用情况

反馈给用户，且软件本身应该足够轻量化，于是我们抛弃了数据库的使用，而是采用了最简单，存储和发布最容易的文本方式，本身我们要展示的就是文本类资源，直接使用文本所耗资源最少，无需额外组件，同时文本读写速度较快。格式化存储的文本数据也可减少 web 端的压力。

2.1.3 数据显示

数据的显示，我们考虑过使用 GUI 来实现，一方面基于 C 语言的 GUI 大多不美观，且使用 C 语言来进行 GUI 的开发并不是一种明智的决定，对开发周期也极其不利，另一方面，系统资源监视器应该尽可能轻量化，加入 GUI 组件会增加软件体积和 CPU 内存等资源的占用，同时对于部分不带显示器，只能通过 SSH 等终端方式来访问的主机来说，使用 GUI 是极其不现实的。故而，我们最终选用了终端显示作为数据展示的方式。

将获得的数据分门别类组织好后，通过终端显示，终端显示使用了 termios 和 ncurses 两个库，前者为 linux 内核提供的 api，后者为第三方库，两者通常一起使用，用于控制终端的输出。

Termios 的缺点在于需要使用转义字符，而 ncurse 提供了与终端无关的字符处理方式，而且可以管理键盘，并且支持多窗体管理，二者同时使用可以完成更多控制操作。

为了使得显示画面较为舒适，我们利用控制字关闭光标显示，移动光标，清除必要行，并进行所需的输出。这种方式一来可以防止滚屏的出现，这可以极大的优化用户的体验，同时由于是通过移动光标擦写来实现显示的，故也不受终端缓冲区的限制。

2.1.4 web 端数据获取

由于数据存储的方式，格式化文本存储方式，web 端极大的减轻了压力，仅需定时读取文本并显示文本即可。

Web 服务器理论上任何一款主流 web 服务器均可，这里我们采用的是 nginx，搭建方便，性能稳定，配置方便。网页的定时部分采用 meta 标签实现，文本的显示使用的是 frame 标签。

对于有着公网 ip 的主机来说，上述配置已经足够，但是对于本地主机来说，却无法通过公网访问到所需资源，这里我们为了将本地主机的端口投射到公网上，实现内网穿透，使用了反向代理的方法，用有着公网 ip 的服务器代理本地主机的端口，这里使用的反向代理服务器是较为主流的 ngrok，ngrok 具有配置方便，GO 语言编写即跨平台较好等优点，利用公网服务器的某一端口反向代理本

地主机的 80 端口，即可让外网访问本地主机的 web 页面，实现外网查看本地主机系统资源使用情况。

2.1.5 为云计算提供支持的费用计算

云计算调度支持模块，主要的功能应包括：对系统资源监视器获得的数据进行二次加工，以转换成我们需要的指标；对以获得指标不断更新记录，并组织成所需的数据结构；当进程结束后，将该数据结构写入到文档中；提供 web 端，可通过 web 访问费用计算结果。

数据的二次加工与标准计费的不同。

对于 cpu 占用时间，传统的系统资源监视器（如 top）计算 cpu 占用时间，是将用户模式运行时间，系统模式运行时间，用户模式 waited-for 进程运行时间和系统模式 waited-for 进程运行时间四者相加后得出，而我们认为这与为云计算调度提供支持有所不合，故而在计算 cpu 占用时间时，我们不再将四者相加，而是将用户模式运行时间和系统模式运行时间相加，这才是进程实际运行在 cpu 上的时间；

对于内存的占用量和交换区占用量，标准计费方式是直接读取 /proc/<pid>/statm 中的数据后显示和记录，但这样只能记录到进程占用的内存和交换区的莫值，无法估计出整个生命周期的内存和交换区实际占用值，故而我们通过对上述设计的资源监视器的数据进行二次加工，通过持续记录进程的内存占用量和交换区占用量，在进程结束时，即可计算出进程的平均内存占用量和平均交换区占用量，这样指标才能替云计算调度提供支持；

对于进程读写，/proc 文件系统，关于读写量的记录有这几个数值，包括请求读入量、请求写入量、系统请求读入量、系统请求写入量、从存储器实际读入量和实际写入存储器的量等数值，所有数值均有内核动态更新和累加，故而此处的关键是选择合适的量作为进程的读写量，标准的计费方式通常选择请求读入写入量即/proc/<pid>/io 中的 rchar 和 wchar 作为数值，这里我们考虑到文件系统、存储设备的不同，认为选择存储设备实际读入写入量作为指标有所不妥，故而和标准的计费方式一样，这里我们也选择了请求读入量和写入量作为进程读写量的指标。

2.2 方案实现

2.2.1 数据获取

Linux 为每一个进程维护一个 task_struct 结构，在结构中可以读取到进程状态，进程标识符，标记符号，优先级等。

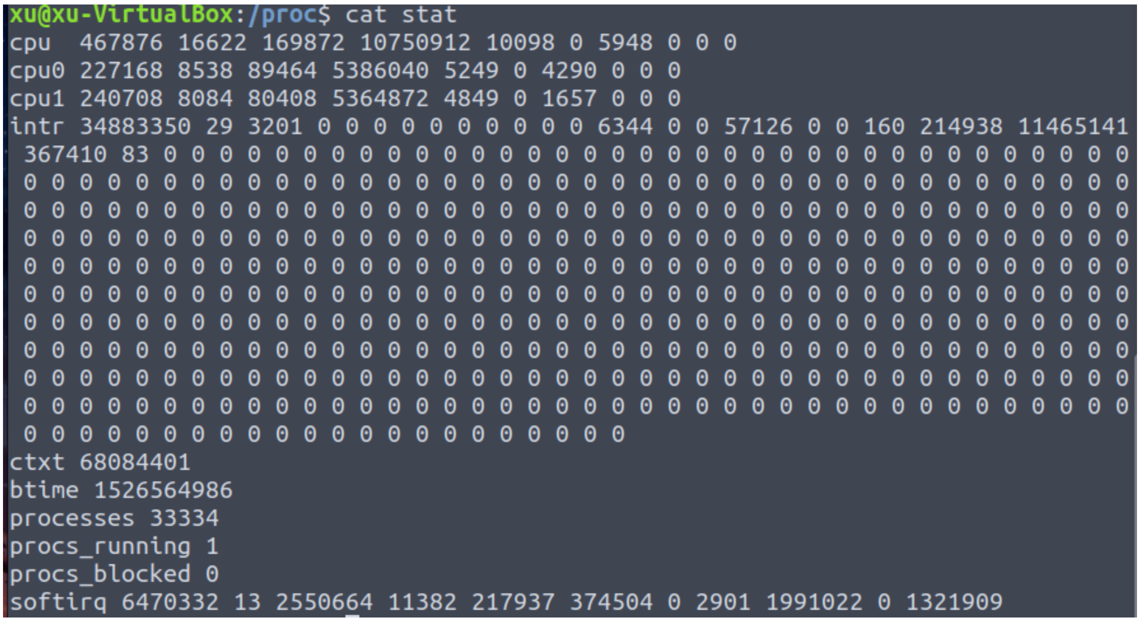
```
pid_t pid;     //进程的标识符
pid_t tgid;    //线程组标识符

int prio, static_prio, normal_prio;
unsigned int rt_priority;
```

但由于全部的进程信息无法由简单的通过进程控制块维护的双向链表遍历得来，所以考虑查询 linux 操作系统是否有一个具体的文件来维护总的进程数量。

Linux 系统维护了一个伪文件系统：/proc 文件系统，这个系统只存在内存中，不占用外存储空间，他以文件的方式为内核和进程提供通讯的接口。值得一提的是这个文件系统中的文件在读取的时候是动态变化的，也就是说这个系统会动态的从内核中取出我们需要的相关信息。

在/proc 下的 stat 中，维护了系统当前运行的全部进程的信息：



```
xu@xu-VirtualBox:/proc$ cat stat
cpu  467876 16622 169872 10750912 10098 0 5948 0 0 0
cpu0 227168 8538 89464 5386040 5249 0 4290 0 0 0
cpu1 240708 8084 80408 5364872 4849 0 1657 0 0 0
intr 34883350 29 3201 0 0 0 0 0 0 0 0 0 0 6344 0 0 57126 0 0 160 214938 11465141
 367410 83 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
ctxt 68084401
btime 1526564986
processes 33334
procs_running 1
procs_blocked 0
softirq 6470332 13 2550664 11382 217937 374504 0 2901 1991022 0 1321909
```

图 2-1 stat 文件的内容

在这个内容中：第一行与 cpu 有关的数据分别是

1．正常的进程在用户态下执行的时间积累

2. NICED 的进程在用户态下执行的时间列

3. 进程在内核态的执行时间积累

4. 空闲时间积累
5. 等待 i/o 完成的时间积累
6. 硬中断时间
7. 软中断时间

再往下的内容中记录了如下的数据

“intr”： 这行给出自系统启动以来的所有中断信息。第一个数字记录所有的中断的次数；然后每个数对应一个特定的中断自系统启动以来所发生的次数。

“ctxt”： 给出了自系统启动以来 CPU 发生的上下文交换的次数。

“btime”： 给出了从系统启动到现在为止的时间，单位为秒。

“processes ”： 自系统启动以来所创建的任务的个数目。

“procs_running”：当前运行队列的任务的数目。

“procs_blocked”：当前被阻塞的任务的数目，等待 I/O 完成次数。

在/proc 目录下，有一些一数字命名的目录，这些目录被称之为进程目录，同

样在这个进程目录下我们可以看到有很多的进程相关的信息。

图 2-2 进程目录里的信息

在进程文件夹里的 stat 文件中可以看到某一个进程里的更多信息

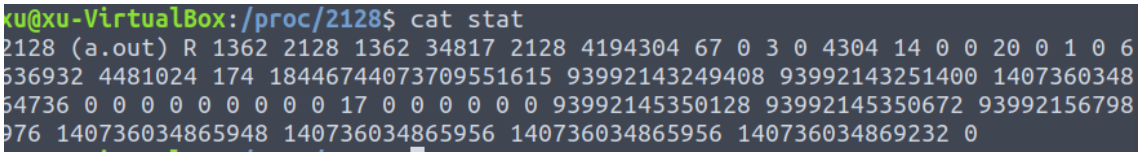


图 2-3 进程 stat 信息

比如刚刚测试运行了一个 a.out，后面一长串信息，查询后面这一大串的字符

信息如下。

pid=2128 进程(包括轻量级进程，即线程)号
comm=a.out 应用程序或命令的名字
task_state=R 任务的状态，R:runnign, S:sleeping (TASK_INTERRUPTIBLE), D:disk sleep (TASK_UNINTERRUPTIBLE), T: stopped, T:tracing stop,Z:zombie, X:dead
ppid=1362 父进程 ID
pgid=2128 线程组号
sid=1362 该任务所在的会话组 ID
tty_nr=34817(pts/3) 该任务的 tty 终端的设备号，INT（34817/256）=主设备号，（34817-主设备号）=次设备号
tty_pgrp=2128 终端的进程组号，当前运行在该任务所在终端的前台任务(包括 shell 应用程序)的 PID。
task->flags=4194303 进程标志位，查看该任务的特性
min_flt=67 该任务不需要从硬盘拷数据而发生的缺页（次缺页）的次数
cmin_flt=0 累计的该任务的所有的 waited-for 进程曾经发生的次缺页的次数目
maj_flt=3 该任务需要从硬盘拷数据而发生的缺页（主缺页）的次数
cmaj_flt=0 累计的该任务的所有的 waited-for 进程曾经发生的主缺页的次数目
utime=4304 该任务在用户态运行的时间，单位为 jiffies
stime=14 该任务在核心态运行的时间，单位为 jiffies
cutime=0 累计的该任务的所有的 waited-for 进程曾经在用户态运行的时间，单位为 jiffies
cstime=0 累计的该任务的所有的 waited-for 进程曾经在核心态运行的时间，单位为 jiffies
priority=20 任务的动态优先级
nice=0 任务的静态优先级
num_threads=1 该任务所在的线程组里线程的个数
it_real_value=0 由于计时间隔导致的下一个 SIGALRM 发送进程的时延，以 jiffy 为单位.
start_time=6636932 该任务启动的时间，单位为 jiffies
vsize=4481024（page） 该任务的虚拟地址空间大小
rss=174 (page) 该任务当前驻留物理地址空间的大小
Number of pages the process has in real memory,minu 3 for administrative

purpose.
这些页可能用于代码，数据和栈。
rlim=18446744073709551615（bytes） 该任务能驻留物理地址空间的最大值
start_code=93992143249408 该任务在虚拟地址空间的代码段的起始地址
end_code=93992143251400 该任务在虚拟地址空间的代码段的结束地址
start_stack=140736034864736 该任务在虚拟地址空间的栈的结束地址
kstkesp=0 esp(32 位堆栈指针) 的当前值, 与在进程的内核堆栈页得到的一致.
kstkeip=2097798 指向将要执行的指令的指针, EIP(32 位指令指针)的当前值.
pendingsig=0 待处理信号的位图，记录发送给进程的普通信号
block_sig=0 阻塞信号的位图
sigign=0 忽略的信号的位图
sigcatch=0 被俘获的信号的位图
wchan=0 如果该进程是睡眠状态，该值给出调度的调用点
nswap 被 swapped 的页数，当前没用
cnswap 所有子进程被 swapped 的页数的和，当前没用
exit_signal=17 该进程结束时，向父进程所发送的信号
task_cpu(task)=0 运行在哪个 CPU 上
task_rt_priority=0 实时进程的相对优先级别
task_policy=0 进程的调度策略，0=非实时进程，1=FIFO 实时进程；2=RR 实时进程

以上的信息我们重点提取出 utime=4304，stime=14，cutime=0，cstime=0 四条，

计算进程总的 cpu 时间为 utime+stime+cutime+cstime。
总的 cpu 使用时间可以使用如下算法。
1、采样两个足够短的时间间隔的 Cpu 快照，分别记作 t1,t2，其中 t1、t2 的结构
均为：(user、nice、system、idle、iowait、irq、softirq、stealstolen、guest)的 9 元组;
2、计算总的 Cpu 时间片 totalCpuTime
a)把第一次的所有 cpu 使用情况求和，得到 s1;
b)把第二次的所有 cpu 使用情况求和，得到 s2;

c)s2 - s1 得到这个时间间隔内的所有时间片，即 totalCpuTime = j2 - j1 ;

3、计算空闲时间 idle

idle 对应第四列的数据，用第二次的 idle - 第一次的 idle 即可

idle=第二次的 idle - 第一次的 idle

4、计算 cpu 使用率

pcpu =100* (total-idle)/total

为了便于使用，我们为每个进程构造一个结构体，用于存储相关信息：

```
typedef struct proc_t {
int tid, ppid;
unsigned pcpu;
char state, pad_1, pad_2, pad_3;
unsigned long long utime, stime, cutime, cstime, start_time;
#ifdef SIGNAL_STRING
Char signal[18], blocked[18], sigignore[18], sigcatch[18], _sigpnd[18];
#else
long long signal, blocked, sigignore, sigcatch, _sigpnd;
#endif
unsigned long long start_code, end_code, start_stack, kstk_esp, kstk_eip, wchan;
long priority, nice, rss, alarm, size, resident, share, trs, lrs, drs, dt;
unsigned long vm_size, vm_lock, vm_rss, vm_data, vm_stack, vm_exe, vm_lib,
rtprio, sched, vsize, rss_rlim, flags, min_flt, maj_flt, cmin_flt, cmaj_flt;
char **environ, **cmdline;
char euser[20], ruser[20], suser[20], fuser[20], rgroup[20], egroup[20], sgroup[20],
fgroup[20], cmd[16];
struct proc_t *ring, *next;
int pgrp, session, nlwp, tgid, tty, euid, egid, ruid, rgid, suid, sgid, fuid, fgid, tpgid,
exit_signal, processor;
}
```

文件信息获取的实现:

这里对于文件的打开和读取，我们没有采用文件指针，而是使用了更为稳健

的文件描述符。

利用 linux 提供的 open 和 read 函数，我们可以分别（打开文件）获得文件描述符和通过文件描述符读取文件内容。

这里以读取/proc/#/stat 为例

```
fd = open(filename, O_RDONLY, 0);
if(fd == -1) return -1;
num_read = read(fd, ret, cap - 1);
close(fd);
```

先将 filename 文件的内容读入到 ret 中。

然后利用 sscanf 将内容赋给对应的变量

```
num = sscanf(S,
    "%c "
    "%d %d %d %d %d "
    "%lu %lu %lu %lu %lu "
    "%Lu %Lu %Lu %Lu "
    "%ld %ld "
    "%d "
    "%ld "
    "%Lu "
    "%lu "
    "%ld "
    "%lu %"KLF"u %"KLF"u %"KLF"u %"KLF"u %"KLF"u "
    "%*s %*s %*s %*s " /* discard, no RT signals & Linux 2.1 used hex */
    "%"KLF"u %*lu %*lu "
    "%d %d "
    "%lu %lu",
    &P->state,
    &P->ppid, &P->pgrp, &P->session, &P->tty, &P->tpgid,
    &P->flags, &P->min_flt, &P->cmin_flt, &P->maj_flt, &P->cmaj_flt,
    &P->utime, &P->stime, &P->cutime, &P->cstime,
    &P->priority, &P->nice,
    &P->nlwp,
```

```
    &P->alarm,
    &P->start_time,
    &P->vsize,
    &P->rss,
    &P->rss_rlim, &P->start_code, &P->end_code, &P->start_stack, &P->kstk_esp,
&P->kstk_eip,
    &P->wchan,
    &P->exit_signal,
    &P->processor,
    &P->rtprio,
    &P->sched
);
```

2.2.2 数据存储

数据存储有两个关键点，一个是文本的构建，二是文本的写入。

C 语言标准库中有一个非常方便的格式化字符串构建函数 sprintf 。

int sprintf(char *buffer, const char *format, [argument] …);

在获取到所需数据后，我们便采用 sprintf 构建我们所需的文本。

```
sprintf(NET.lo_bytes_receive_s, "%lldGB", NET.lo_bytes_receive / GB);
```

为了实现部分封装的特性，我们利用 C 语言的可变参数将部分构造操作封装为了函数。

```
static const char *FormatMake (const char *fmts, ...)
{
    va_list va;

    va_start(va, fmts);
    vsnprintf(_buf, sizeof(buf), fmts, va);
    va_end(va);
    return (const char *)_buf;
}
```

为了配合可变参数的使用，这里采用 vsnprintf 来格式化构造字符串。

文件的写入，我们并没有采用 c 语言标准库提供的 file 指针，而是使用了更

为稳定可靠的文件描述符进行操作，使用 linux 提供的 api 完成文件的打开，清空，写入，指针的移动等。

首先是利用 open 函数以只写方式打开所需文件

```
txt_fd = open("/usr/local/nginx/html/test/test.txt", O_WRONLY, 0);
```

然后在每一次的写入前，首先要清空该文件

```
ftruncate(txt_fd, 0);
```

然后利用 lseek 函数移动指针到文件头

```
lseek(txt_fd, 0, SEEK_SET);
```

接下来仅需使用 write 函数利用文件描述符写入要提供给 web 端的数据即可。

```
write(txt_fd, NET_TITLE, strlen(NET_TITLE));
```

由于写入的数据均为字符串，故而采用了 strlen 函数来计算数据长度。

2.2.3 数据显示

每次程序启动时，首先是要利用 termios 提供的 api 设置当前终端属性

```
setupterm(NULL, STDOUT_FILENO, NULL);
```

紧接着利用 curses 提供的接口来完成必要的操作，包括：

1. 清屏

```
putp(clear_screen);
```

2. 指针不显示

```
putp(cursor_invisible);
```

3. 指针移动

```
putp(tgoto(cursor_address, 0, 3));
```

这里利用了当前指针地址、目的地址和 tgoto 函数计算出指针所需的动作，并将其转换为命令字后有 putp 输入到终端。

4. 清除当前位置到行尾

```
putp(clr_eol);
```

2.2.4 web 端

由于数据的格式化存储，使得 web 端实现极其容易，此次采用的是 nginx 做为 web 服务器，使用 ngrok 作为反向代理。

Web 端使用了 meta 标签，设置 3 秒刷新一次

```
<meta http-equiv="refresh"content="3">
```

使用 frame 标签来获取并显示相关文本

<frame src="http://www.thelock.site/test/test.txt"name="ifram"></frame>

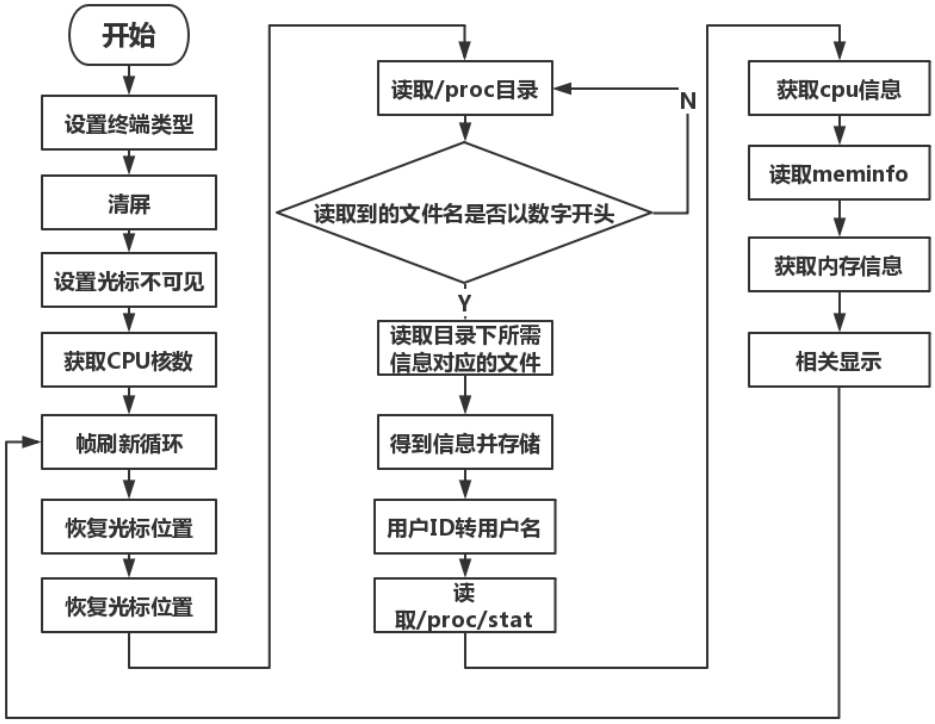


图 2-4 总流程图

2.2.5 为云计算调度提供支持的费用计算

在标准计费中，我们无需存储每一次读取到的数据，但是在为云计算调度提供支持的费用计算时，我们需要一个合适的数据结构来存储和更新每一个读取到的数据。

```
typedef struct process_t{
    char cmd[21];
    unsigned cput;//jiffies
    unsigned long memt;//kb
    unsigned long swap;//kb
    char rchar[10], wchar[10];
    struct process_t *next;
    unsigned long seconds;
    unsigned char alive;
}process_t, *process_p;
```

图 2-5 process 数据结构

这里我们采用的是最传统的链表方式，优点是节省内存。这里我们用到了一个 alive 标志位，当一个更新过后，如果某个进程依然存在，该标志为会被置为 1，反之置为 0，表示该进程已经结束，需要将该进程的数据记录到库中。

数据结构中的 cput 是/proc/<pid>/stat 中的 utime 和 stime 之和（而标准计费是 utime,stime,cutime 和 cstime 之和）。memt 是每秒读取/proc 时读取到的内存占用量之和（常驻内存量），swap 是每秒读取/proc 时读取到的交换区占用量之和，seconds 是统计该进程的总时间，用来计算平均内存占用量和平均交换区占用量。rchar 和 wchar 则是采取的和标准方式同样的读取方式，直接读取的/proc/<pid>/io 中的 rchar 和 wchar。

得到我们所需要的数据后，我们只需不断更新该数据结构，直到某次读取完毕后，某进程的 alive 标志位未被置为 1，则我们将该进程的数据格式化后，放入到用于存储结束进程信息的文本中去。

```
sprintf(process_buf,"cmd:%s\tcpu:%ujiffies\tmem(average):%ldKB\tswap(average):%ldKB\tread
:%s\twrite:%s\n", process->next->cmd, process->next->cput, process->next->memt /
process->next->seconds, process->next->swap / process->next->seconds, process->next->rchar,
process->next->wchar);
write(proc_fd, process_buf, strlen(process_buf));
```

对于存储信息的文本，我们为了便于从 web 端访问，采用了类似系统资源监视器的处理方式。

```
<html>
  <head>
    <meta http-equiv="refresh"content="3">
  </head>
  <frameset>
    <frame src="http://www.thelock.site/test/proc.txt"name="iframe"></frame>
  </frameset>
</html>
```

第三章 存在问题与解决方案

3.1 存在的主要问题

在输出结果到命令行的时候，显示的计费数据滚动，可视性与交互性都非常的差。这里希望能实现一个刷新屏幕的功能。

在完成 web 端的搭建和编写后，在本地主机测试时，我们意识到只有局域网的终端上才能访问到测试主机的 web 页面。

3.2 解决方案

（针对发现的问题，通过分析文献寻求可替代的解决方案，并说明拟采取解决方案相对于其他解决方案的优势。）

无疑我们要处理的是字符终端，linux 提供给字符终端处理的有 ncurses 库。

```
sudo apt-get install libncurses5-dev
```

ncurses 库中，提供了函数”setupterm”用于设置终端类型，这里我们使用了

```
setupterm(NULL, STDOUT_FILENO, NULL);
```

当第一个参数为 NULL 时，使用环境变量 TERM 的值，终端类型用来查找相应的数据库以获得信息。

我们可以使用 string capabilities 来对终端进行控制，如

```
putp(clear_screen);
putp(cursor_invisible);
```

其中 clear_screen 用于清屏，cursor_ivisible 用于设置光标不可见，其余的 cap 可在 man 5 terminfo 中查到。

通过 ncurses 库提供的 tgoto 函数和 cursor_address 我们可以得到控制光标移动的转义序列。

```
putp(tgoto(cursor_address, 0, 3));
```

上述函数可以将光标移动到 3 行 0 列处。

于是通过光标移动转义序列、清空行剩余列、清空行等转义序列，我们便可以达到不滚动同时刷新终端的目的。

我们在网络上查找了很多关于内网穿透的资料，最后选择了目前较为主流的

一款内网穿透工具 ngrok。

```
wget https://github.com/inconshreveable/ngrok/archive/1.7.1.tar.gz
```

```
tar -xvf 1.7.1.tar.gz
```

ngrok 的可以分别为客户端和服务器端编译，由于采用 GO 语言编写，可以轻松的实现跨平台编译。

服务器端编译

```
./configure && make release-server
```

以编译 arm 处理器的 linux 平台客户端为例

```
GOARCH=arm GOOS=linux make release-client
```

客户端的配置文件也比较通俗易用

```
server_addr: "ngrok.thelock.site:4443"
trust_host_root_certs: false
tunnels:
  http:
    remote_port: 8888
    proto:
      http: 80
```

在打开 ngrok 服务器后，客户端仅需一条指令即可实现反向代理

```
sudo ./ngrok -config ngrok.conf start http
```

第四章 执行情况与完成度

在实现了基础的系统资源监视器功能的基础上，进行了 web 端的开发和内网穿透。

主要任务全部完成，包括：在 Linux 系统下，监控显示每一个进程的 CPU 使用量、内存使用量、交换内存、缓存大小、缓冲区大小、流程 PID、用户、命令等信息，用程序实现针对上面指标的显示列表、计费，用于控制特别进程的使用，为云计算调度，提供支持。

完成了进程总数目，各状态进程数目，cpu 占有量，内存总量、使用量、剩余量，交换区总量、使用量、剩余量，用于内核缓冲的内存总量和用于缓存的交换区总量，进程 CPU 使用量、内存使用量、内存使用率、交换内存、PID、父进程 ID、real user、所有者、real user grou、所有者 group、网络设备流量、进程 IO 数据等数据的读取和格式化存储。

完成了字符终端页面的绘制、刷新等。

完成了 web 端环境的搭建，包括 web 服务器和 ngrok 等，完成了简单网页的编写。

第五章 分工协作与交流情况

王铁举，许奕腾：负责 linux 平台上系统资源监视器的代码编写和报告编写。

刁瑞琪，杨浩铨：负责 web 端程序的开发，web 服务器和 ngrok 的调试，以及 PPT 的制作。

徐智林，邓巧：负责服务器环境搭建和本地主机测试。

参考文献

[1] linux 状态及原理全剖析：倾枫斜阳

[2] SLinux curses 库使用：byxdaz

致谢

本报告的工作是在我的指导教师文军老师的悉心指导下完成的，非常感谢老师的指导。