



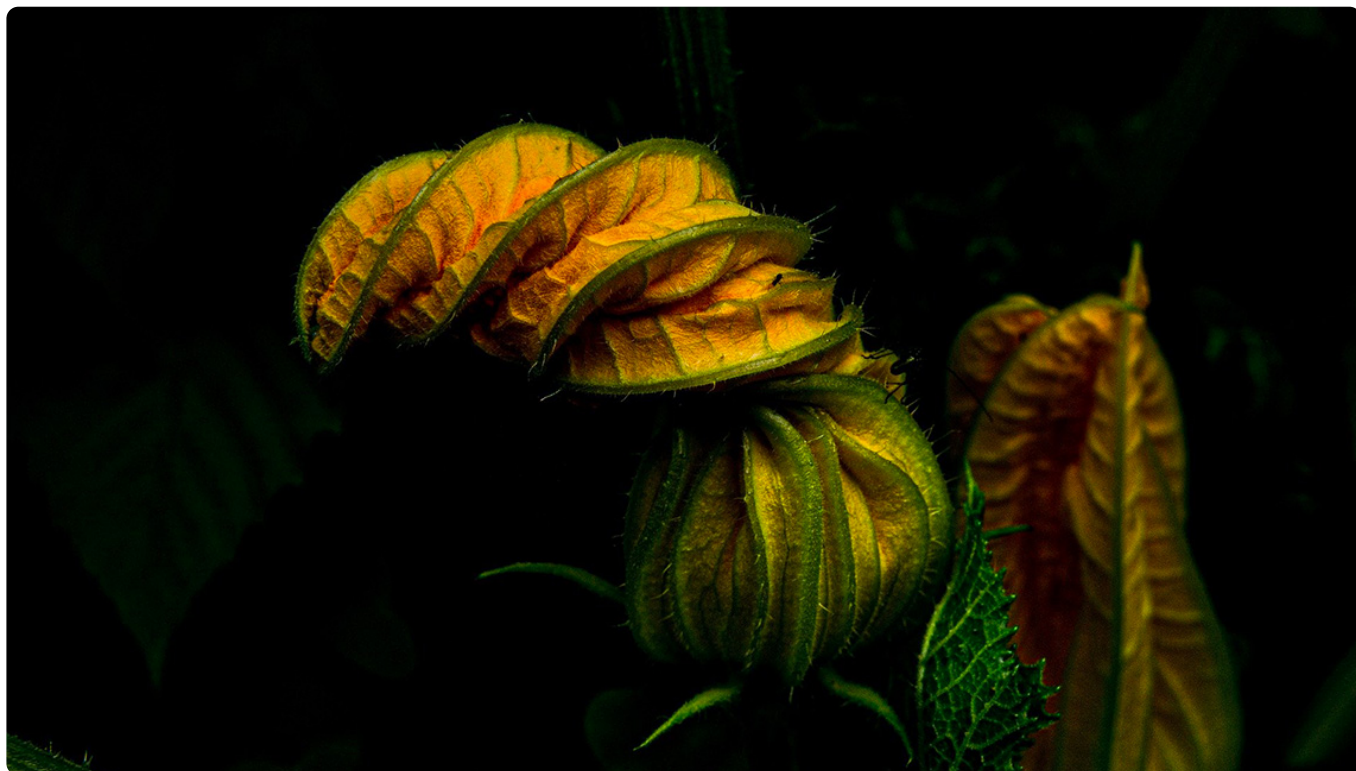
下载APP



20 分析篇 | 如何分析CPU利用率飙升问题？

2020-10-03 邵亚方

Linux内核技术实战课

[进入课程 >](#)**讲述：邵亚方**

时长 14:41 大小 13.46M



你好，我是邵亚方。

如果你是一名应用开发者，那你应该知道如何去分析应用逻辑，对于如何优化应用代码提升系统性能也应该有自己的一套经验。而我们这节课想要讨论的是，如何拓展你的边界，让你能够分析代码之外的模块，以及对你而言几乎是黑盒的 Linux 内核。

在很多情况下，应用的性能问题都需要通过分析内核行为来解决，因此，内核提供了非常多的指标供应用程序参考。当应用出现问题时，我们可以查看到底是哪些指标出现了异常，然后再做进一步分析。不过，这些内核导出的指标并不能覆盖所有的场景，我们面临的问题可能更加棘手：应用出现性能问题，可是系统中所有的指标都看起来没有异常。相信很多人都为此抓狂过。那出现这种情况时，内核到底有没有问题呢，它究竟在搞什么鬼？这节课我就带你探讨一下如何分析这类问题。



我们知道，对于应用开发者而言，应用程序的边界是系统调用，进入到系统调用中就是 Linux 内核了。所以，要想拓展分析问题的边界，你首先需要知道该怎么去分析应用程序使用的系统调用函数。对于内核开发者而言，边界同样是系统调用，系统调用之外是应用程序。如果内核开发者想要拓展分析问题的边界，也需要知道如何利用系统调用去追踪应用程序的逻辑。

如何拓展你分析问题的边界？

作为一名内核开发者，我对应用程序逻辑的了解没有对内核的了解那么深。不过，当应用开发者向我寻求帮助时，尽管我对他们的应用逻辑一无所知，但这并不影响我对问题的分析，因为我知道如何借助分析工具追踪应用程序的逻辑。经过一系列追踪之后，我就能对应用程序有一个大概的认识。

我常用来追踪应用逻辑的工具之一就是 strace。strace 可以用来分析应用和内核的“边界”——系统调用。借助 strace，我们不仅能够了解应用执行的逻辑，还可以了解内核逻辑。那么，作为应用开发者的你，就可以借助这个工具来拓展你分析应用问题的边界。

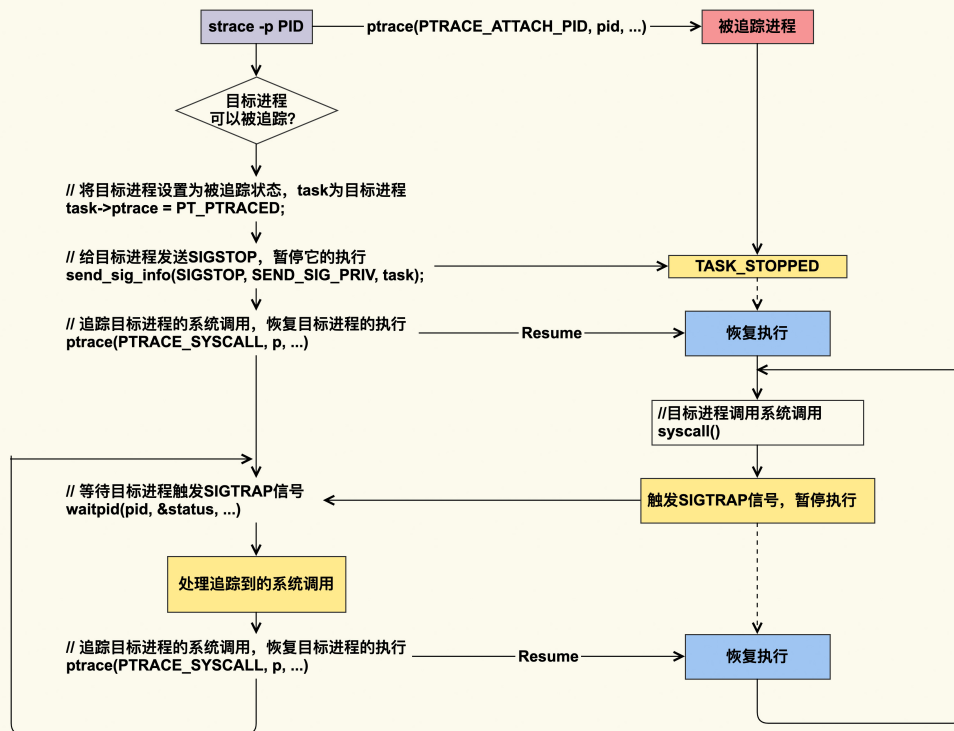
strace 可以跟踪进程的系统调用、特定的系统调用以及系统调用的执行时间。很多时候，我们通过系统调用的执行时间，就能判断出业务延迟发生在哪里。比如我们想要跟踪一个多线程程序的系统调用情况，那就可以这样使用 strace：

```
$ strace -T -tt -ff -p pid -o strace.out
```

不过，在使用 strace 跟踪进程之前，我希望你可以先明白 strace 的工作原理，这也是我们这节课的目的：你不只要知道怎样使用工具，更要明白工具的原理，这样在出现问题时，你就能明白该工具是否适用了。

了解工具的原理，不要局限于如何使用它

strace 工具的原理如下图所示（我们以上面的那个命令为例来说明）：



strace基本原理

我们从图中可以看到，对于正在运行的进程而言，strace 可以 attach 到目标进程上，这是通过 ptrace 这个系统调用实现的（gdb 工具也是如此）。ptrace 的 PTRACE_SYSCALL 会去追踪目标进程的系统调用；目标进程被追踪后，每次进入 syscall，都会产生 SIGTRAP 信号并暂停执行；追踪者通过目标进程触发的 SIGTRAP 信号，就可以知道目标进程进入了系统调用，然后追踪者会去处理该系统调用，我们用 strace 命令观察到的信息输出就是该处理的结果；追踪者处理完该系统调用后，就会恢复目标进程的执行。被恢复的目标进程会一直执行下去，直到下一个系统调用。

你可以发现，目标进程每执行一次系统调用都会被打断，等 strace 处理完后，目标进程才能继续执行，这就会给目标进程带来比较明显的延迟。因此，在生产环境中我不建议使用该命令，如果你要使用该命令来追踪生产环境的问题，那就一定要做好预案。

假设我们使用 strace 跟踪到，线程延迟抖动是由某一个系统调用耗时长导致的，那么接下来我们该怎么继续追踪呢？这就到了应用开发者和运维人员需要拓展分析边界的时刻了，对内核开发者来说，这才算是分析问题的开始。

学会使用内核开发者常用的分析工具

我们以一个实际案例来说明吧。有一次，业务开发者反馈说他们用 `strace` 追踪发现业务的 `pread(2)` 系统调用耗时很长，经常会有几十毫秒（ms）的情况，甚至能够达到秒级，但是不清楚接下来该如何分析，因此让我帮他们分析一下。

因为已经明确了问题是由 `pread(2)` 这个系统调用引起的，所以对内核开发者而言，后续的分析就相对容易了。分析这类问题最合适的工具是 `ftrace`，我们可以使用 `ftrace` 的 `function_trace` 功能来追踪 `pread(2)` 这个系统调用到底是在哪里耗费了这么长的时间。

要想追踪 `pread(2)` 究竟在哪里耗时长，我们就需要知道该系统调用对应的内核函数是什么。我们有两种途径可以方便地获取到系统调用对应的内核函数：

查看 [include/linux/syscalls.h](#) 文件里的内核函数：

你可以看到，与 `pread` 有关的函数有多个，由于我们的系统是 64bit 的，只需关注 64bit 相关的系统调用就可以了，所以我们锁定在 `ksys_pread64` 和 `sys_read64` 这两个函数上。[通过该头文件里的注释](#)我们能知道，前者是内核使用的，后者是导出给用户的。那在内核里，我们就需要去追踪前者。另外，请注意，不同内核版本对应的函数可能不一致，我们这里是以最新内核代码 (5.9-rc) 为例来说明的。

通过 `/proc/kallsyms` 这个文件来查找：

```
$ cat /proc/kallsyms | grep pread64
...
ffffffffa02ef3d0 T ksys_pread64
...
```

`/proc/kallsyms` 里的每一行都是一个符号，其中第一列是符号地址，第二列是符号的属性，第三列是符号名字，比如上面这个信息中的 `T` 就表示全局代码符号，我们可以追踪这类的符号。关于这些符号属性的含义，你可以通过 [man nm](#) 来查看。

接下来我们就使用 `ftrace` 的 `function_graph` 功能来追踪 `ksys_pread64` 这个函数，看看究竟是内核的哪里耗时这么久。`function_graph` 的使用方式如下：

```

2 # 首先设置要追踪的函数
3 $ echo ksys_pread64 > /sys/kernel/debug/tracing/set_graph_function
4
5 # 其次设置要追踪的线程的pid, 如果有多个线程, 那需要将每个线程都逐个写入
6 $ echo 6577 > /sys/kernel/debug/tracing/set_ftrace_pid
7 $ echo 6589 >> /sys/kernel/debug/tracing/set_ftrace_pid
8
9 # 将function_graph设置为当前的tracer, 来追踪函数调用情况
$ echo function_graph > /sys/kernel/debug/tracing/current_tracer

```

然后我们就可以通过 `/sys/kernel/debug/tracing/trace_pipe` 来查看它的输出了, 下面就是我追踪到的耗时情况:

```

21)      io_schedule() {
21)      io_schedule_timeout() {
21)      __delayacct_blkio_start() {
21)      0.107 us      ktime_get_ts64();
21)      0.456 us      }
21)      schedule_timeout() {
21)      schedule() {
21)      __schedule() {
21)      0.036 us      rcu_note_context_switch();
21)      0.071 us      _raw_spin_lock_irq();
21)      deactivate_task() {
21)      dequeue_task() {
21)      0.075 us      update_rq_clock.part.79();
21)      dequeue_task_fair() {
21)      dequeue_entity() {
21)      update_curr() {
21)      0.051 us      update_min_vruntime();
21)      0.066 us      cpuacct_charge();
21)      0.928 us      }
21)      0.082 us      update_cfs_rq_blocked_load();
21)      0.036 us      clear_buddies();
21)      0.053 us      account_entity_dequeue();
21)      0.035 us      update_min_vruntime();
21)      0.043 us      update_cfs_shares();
21)      3.202 us      }
21)      0.036 us      hrtick_update();
21)      3.950 us      }
21)      4.708 us      }
21)      5.063 us      }
21)      idle_balance() {
21)      msecs_to_jiffies();
21)      0.037 us      }
21)      0.505 us      put_prev_task_fair();
21)      0.073 us      pick_next_task_fair();
21)      0.038 us      pick_next_task_idle();
21)      0.037 us
-----
21) worker--6577 => worker--6589
-----

21)      0.164 us      finish_task_switch();
21) ! 102950.8 us      }

```

我们可以发现 `pread(2)` 有 102ms 是阻塞在 `io_schedule()` 这个函数里的, `io_schedule()` 的意思是, 该线程因 I/O 阻塞而被调度走, 线程需要等待 I/O 完成才能继续执行。在

function_graph 里，我们同样也能看到 **pread** (2)**** 是如何一步步执行到 io_schedule 的，由于整个流程比较长，我在这里只把关键的调用逻辑贴出来：

[复制代码](#)

```
1  21)          |          __lock_page_killable() {
2  21)    0.073 us |          page_waitqueue();
3  21)          |          __wait_on_bit_lock() {
4  21)          |          prepare_to_wait_exclusive() {
5  21)    0.186 us |          _raw_spin_lock_irqsave();
6  21)    0.051 us |          _raw_spin_unlock_irqrestore();
7  21)    1.339 us |          }
8  21)          |          bit_wait_io() {
9  21)          |          io_schedule() {
```

我们可以看到，**pread (2)** 是从 `__lock_page_killable` 这个函数调用下来的。当 `pread(2)` 从磁盘中读文件到内存页 (page) 时，会先 lock 该 page，读完后再 unlock。如果该 page 已经被别的线程 lock 了，比如在 I/O 过程中被 lock，那么 `pread(2)` 就需要等待。等该 page 被 I/O 线程 unlock 后，`pread(2)` 才能继续把文件内容读到这个 page 中。我们当时遇到的情况是：在 `pread(2)` 从磁盘中读取文件内容到一个 page 中的时候，该 page 已经被 lock 了，于是调用 `pread(2)` 的线程就在这里等待。这其实是合理的内核逻辑，没有什么问题。接下来，我们就需要看看为什么该 page 会被 lock 了这么久。

因为线程是阻塞在磁盘 I/O 里的，所以我们需要查看一下系统的磁盘 I/O 情况，我们可以使用 `iostat` 来观察：

```
$ iostat -dxm 1
```

追踪信息如下：

Device:	rrqm/s	wrqm/s	r/s	w/s	rMB/s	wMB/s	avgrq-sz	avgqu-sz	await	r_await	w_await	svctm	%util
sdb	0.00	0.00	855.00	340.00	63.31	80.44	246.35	0.32	0.27	0.28	0.24	0.15	18.30
sda	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Device:	rrqm/s	wrqm/s	r/s	w/s	rMB/s	wMB/s	avgrq-sz	avgqu-sz	await	r_await	w_await	svctm	%util
sdb	0.00	0.00	1138.00	337.00	69.06	83.21	211.42	0.54	0.36	0.32	0.52	0.17	25.50
sda	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Device:	rrqm/s	wrqm/s	r/s	w/s	rMB/s	wMB/s	avgrq-sz	avgqu-sz	await	r_await	w_await	svctm	%util
sdb	0.00	0.00	997.00	1230.00	67.64	301.17	339.17	3.33	1.19	1.98	0.54	0.26	58.20
sda	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Device:	rrqm/s	wrqm/s	r/s	w/s	rMB/s	wMB/s	avgrq-sz	avgqu-sz	await	r_await	w_await	svctm	%util
sdb	0.00	0.00	576.00	2524.00	14.48	622.43	420.77	15.83	5.20	26.26	0.39	0.32	100.00
sda	0.00	1.00	0.00	6.00	0.00	0.04	13.33	0.00	0.17	0.00	0.17	0.17	0.10

Device:	rrqm/s	wrqm/s	r/s	w/s	rMB/s	wMB/s	avgrq-sz	avgqu-sz	await	r_await	w_await	svctm	%util
sdb	0.00	0.00	631.00	534.00	16.11	129.69	256.31	14.00	12.36	22.41	0.49	0.81	94.30
sda	0.00	0.00	0.00	11.00	0.00	0.05	9.45	0.00	0.00	0.00	0.00	0.00	0.00

Device:	rrqm/s	wrqm/s	r/s	w/s	rMB/s	wMB/s	avgrq-sz	avgqu-sz	await	r_await	w_await	svctm	%util
sdb	0.00	0.00	1280.00	672.00	94.68	162.77	270.11	1.13	0.58	0.63	0.49	0.23	44.00
sda	0.00	0.00	0.00	7.00	0.00	0.25	73.14	0.00	0.00	0.00	0.00	0.00	0.00

Device:	rrqm/s	wrqm/s	r/s	w/s	rMB/s	wMB/s	avgrq-sz	avgqu-sz	await	r_await	w_await	svctm	%util
sdb	0.00	0.00	1554.00	653.00	99.50	161.52	242.21	1.46	0.66	0.73	0.51	0.25	55.90
sda	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00


其中，sdb 是业务 pread(2) 读取的磁盘所在的文件，通常情况下它的读写量很小，但是我们从上图中可以看到，磁盘利用率（%util）会随机出现比较高的情况，接近 100%。而且 avgrq-sz 很大，也就是说出现了很多 I/O 排队的情况。另外，w/s 比平时也要高很多。我们还可以看到，由于此时存在大量的 I/O 写操作，磁盘 I/O 排队严重，磁盘 I/O 利用率也很高。根据这些信息我们可以判断，之所以 pread(2) 读磁盘文件耗时较长，很可能是因为被写操作饿死导致的。因此，我们接下来需要排查到底是谁在进行写 I/O 操作。

通过 iotop 观察 I/O 行为，我们发现并没有用户线程在进行 I/O 写操作，写操作几乎都是内核线程 kworker 来执行的，也就是说用户线程把内容写在了 Page Cache 里，然后 kworker 将这些 Page Cache 中的内容再同步到磁盘中。这就涉及到了我们这门课程第一个模块的内容了：如何观测 Page Cache 的行为。

自己写分析工具

如果你现在还不清楚该如何来观测 Page Cache 的行为，那我建议你再从开头仔细看一遍我们这门课程的第一个模块，我在这里就不细说了。不过，我要提一下在 Page Cache 模块中未曾提到的一些方法，这些方法用于判断内存中都有哪些文件以及这些文件的大小。

常规方式是用 fincore 和 mincore，不过它们都比较低效。这里有一个更加高效的方式：通过写一个内核模块遍历 inode 来查看 Page Cache 的组成。该模块的代码较多，我只说一下核心的思想，伪代码大致如下：

 复制代码

```
1 iterate_supers // 遍历super block
2   iterate_pagecache_sb // 遍历superblock里的inode
```

```
3      list_for_each_entry(inode, &sb->s_inodes, i_sb_list)
4          // 记录该inode的pagecache大小
5          nrpages = inode->i_mapping->nrpages;
6          /* 获取该inode对应的dentry, 然后根据该dentry来查找文件路径;
7           * 请注意inode可能没有对应的dentry, 因为dentry可能被回收掉了,
8           * 此时就无法查看该inode对应的文件名了。
9           */
10         dentry = dentry_from_inode(inode);
11         dentry_path_raw(dentry, filename, PATH_MAX);
```

使用这种方式不仅可以查看进程正在打开的文件，也能查看文件已经被进程关闭，但文件内容还在内存中的情况。所以这种方式分析起来会更全面。

通过查看 Page Cache 的文件内容，我们发现某些特定的文件占用的内存特别大，但是这些文件都是一些离线业务的文件，也就是不重要业务的文件。因为离线业务占用了大量的 Page Cache，导致该在线业务的 workingset 大大减小，所以 `pread(2)` 在读文件内容时经常命中不了 Page Cache，进而需要从磁盘来读文件，也就是说该在线业务存在大量的 pagein 和 pageout。

至此，问题的解决方案也就有了：我们可以通过限制离线业务的 Page Cache 大小，来保障在线业务的 workingset，防止它出现较多的 refault。经过这样调整后，业务再也没有出现这种性能抖动了。

你是不是对我上面提到的这些名字感到困惑呢？也不清楚 inode 和 Page Cache 是什么关系？如果是的话，那就说明你没有好好学习我们这门课程的 Page Cache 模块，我建议你从头再仔细学习一遍。

好了，我们这节课就讲到这里。

课堂总结

我们这节课的内容，对于应用开发者和运维人员而言是有些难度的。我之所以讲这些有难度的内容，就是希望你可以拓展分析问题的边界。这节课的内容对内核开发者而言基本都是基础知识，如果你看不太明白，说明你对内核的理解还不够，你需要花更多的时间好好学习它。我研究内核已经有很多年了，尽管如此，我还是觉得自己对它的理解仍然不够深刻，需要持续不断地学习和研究，而我也一直在这么做。

我们现在回顾一下这节课的重点：

strace 工具是应用和内核的边界，如果你是一名应用开发者，并且想去拓展分析问题的边界，那你就需要去了解 strace 的原理，还需要了解如何去分析 strace 发现的问题；

fttrace 是分析内核问题的利器，你需要去了解它；

你需要根据自己的问题来实现特定的问题分析工具，要想更好地实现这些分析工具，你必须掌握很多内核细节。

课后作业

关于我们这节课的“自己写分析工具”这部分，我给你留一个作业，这也是我没有精力和时间去做的一件事：请你在 sysrq 里实现一个功能，让它可以显示出系统中所有 R 和 D 状态的任务，以此来帮助开发者分析系统 load 飙升的问题。

我在我们的内核里已经实现了该功能，不过在推给 Linux 内核时，maintainer 希望我可以用另一种方式来实现。由于那个时候我在忙其他事情，这件事便被搁置了下来。如果你实现得比较好，你可以把它提交给 Linux 内核，提交的时候你也可以 cc 一下我

(laoar.shao@gmail.com)。对了，你在实现时，也可以参考我之前的提交记录：

[🔗 scheduler: enhancement to show_state_filter and SysRq](#)。欢迎你在留言区与我讨论。

最后，感谢你的阅读，如果你认为这节课的内容有收获，也欢迎把它分享给你的朋友。

提建议

更多课程推荐

数据结构与算法之美

为工程师量身打造的数据结构与算法私教课

王争

前 Google 工程师



立省 ¥40

破 90000 订阅特惠，到手价 ¥89

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 19 案例篇 | 网络吞吐高的业务是否需要开启网卡特性呢？

下一篇 加餐 | 我是如何使用tracepoint来分析内核Bug的？

精选留言 (2)

写留言



Geek_9bf0b0

2020-10-10

邵老师，关于在 sysrq 里实现显示出系统中所有 R 和 D 状态的任务的功能，我的想法是

show_state_filter()接口的state_filter参数改成指针类型，传递参数NULL时表示显示所有进程信息，...

展开



我来也

2020-10-03

老师文中提到的`查看 Page Cache 的组成`这个功能,感觉很吸引人啊!

根据老师的提示,也只找到了这两个方式:

[Is it possible to list the files that are cached?](https://serverfault.com/a/782640)...

展开 ✓

作者回复: 这个方法需要修改内核来实现, 或者写一个内核模块来实现。 主要思路我已经写在文章里了, 你可以思考下如何实现。

