**Memory inside Linux containers**

Or *why don’t*free*and*top*work in a Linux container*?

Lately at Heroku, we have been trying to find the best way to expose memory usage and limits inside Linux containers. It would be easy to do it in a vendor-specific way: most container specific metrics are available at the [cgroup filesystem](https://www.kernel.org/doc/Documentation/cgroups/memory.txt) via /path/to/cgroup/memory.stat, /path/to/cgroup/memory.usage\_in\_bytes, /path/to/cgroup/memory.limit\_in\_bytes and others.

An implementation of Linux containers could easily inject one or more of those files inside containers. Here is an hypothetical example of what Heroku, Docker and others could do:

|  |
| --- |
| # create a new dyno (container):  $ heroku run bash    # then, inside the dyno:  (dyno) $ cat /sys/fs/cgroup/memory/memory.stat  cache 15582273536  rss 2308546560  mapped\_file 275681280  swap 94928896  pgpgin 30203686979  pgpgout 30199319103  # ... |

/sys/fs/cgroup/ is the [recommended location](http://www.freedesktop.org/wiki/Software/systemd/PaxControlGroups/#generalrecommendations) for cgroup hierarchies, but it is **not a standard**. If a tool or library is trying to read from it, and be portable across multiple container implementations, it would need to discover the location first by parsing /proc/self/cgroup and /proc/self/mountinfo. Further, /sys/fs/cgroup is just an umbrella for all cgroup hierarchies, there is no recommendation or standard for *my own cgroup location*. Thinking about it, /sys/fs/cgroup/self would not be a bad idea.

If we decide to go down that path, I would personally prefer to work with the rest of the Linux containers community first and come up with a standard.

I wish it were that simple.

**The problem**

Most of the Linux tools providing system resource metrics were created before cgroups even existed (e.g.: free and top, both from [procps](http://procps.sourceforge.net/)). They usually read memory metrics from the proc filesystem: /proc/meminfo, /proc/vmstat, /proc/PID/smaps and others.

Unfortunately /proc/meminfo, /proc/vmstat and friends are not *containerized*. Meaning that they are not *cgroup-aware*. They will always display memory numbers from the host system (physical or virtual machine) as a whole, which is useless for modern Linux containers (Heroku, Docker, etc.). Processes inside a container can not rely on free, top and others to determine how much memory they have to work with; they are subject to limits imposed by their cgroups and can’t use all the memory available in the host system.

This causes a [lot of confusion](https://bugs.launchpad.net/ubuntu/+source/lxc/+bug/1166730) for users of Linux containers. Why does free say there is 32GB free memory, when the container only allows 512MB to be used?

With the popularization of linux container technologies – [Heroku](https://www.heroku.com/), [Docker](https://www.docker.io/), [LXC](https://linuxcontainers.org/) (version [1.0 was recently released](https://www.stgraber.org/2013/12/20/lxc-1-0-blog-post-series/)), [CoreOS](https://coreos.com/), [lmctfy](https://github.scom/google/lmctfy), [systemd](http://www.freedesktop.org/wiki/Software/systemd/ControlGroupInterface/) and friends – more and more people will face the same problem. It is time to start fixing it.

**Why is this important?**

Visibility into memory usage is very important. It allows people running applications inside containers to optimize their code and troubleshoot problems: memory leaks, swap space usage, etc.

Some time ago, we shipped [log-runtime-metrics](https://devcenter.heroku.com/articles/log-runtime-metrics#memory-swap) at Heroku, as an experimental labs feature. It is not a portable solution though, and does not expose the information inside containers, so that monitoring agents could read it. To make things worse, most of the monitoring agents out there (e.g.: NewRelic?[1](https://fabiokung.com/2014/03/13/memory-inside-linux-containers/#fn-433-1)) rely on information provided by free, /proc/meminfo, etc. That is plain broken inside Linux containers.

On top of that, more and more people have been trying to maximize resource usage inside containers, usually by auto-scaling the number of workers, processes or threads running inside them. This is usually a function of how much memory is available (and/or free) inside the container, and for that do be done programmatically, the information needs to be accessible from inside the container.

**More about**/proc

In case you wondered, none of the files provided by the cgroup filesystem (/sys/fs/cgroup/memory/memory.\*) can be used as a drop-in replacement (i.e.: bind mounted on top of) for /proc/meminfo, or /proc/vmstat. They have different formats and use slightly different names for each metric. Why memory.stat and friends decided to use a format different from what was already being used at /proc/meminfo is beyond my comprehension.

Some of the contents of a /proc filesystem are properly containerized, like the /proc/PID/\* and /proc/net/\* namespaces, but not all of them. Unfortunately, /proc in general is considered to be a mess. From the excellent “[*Creating Linux virtual filesystems*](http://lwn.net/Articles/57369/)” article on [LWN](http://lwn.net/):

*Linus and numerous other kernel developers dislike the ioctl() system call, seeing it as an uncontrolled way of adding new system calls to the kernel. Putting new files into /proc is also discouraged, since that area is seen as being a bit of a mess. Developers who populate their code with ioctl() implementations or /proc files are often encouraged to create a standalone virtual filesystem instead.*

I went ahead and started experimenting with that: [procg](https://github.com/fabiokung/procg/) is an alternative procfilesystem that can be mounted inside linux containers. It replaces /proc/meminfo with a version that reads cgroup specific information. My goal was for it to be a drop-in replacement for proc, without requiring any patches to the Linux kernel. Unfortunately, I later found that [it was not possible](https://github.com/fabiokung/procg/tree/master/patches), because none of the functions to read memory statistics from a cgroup ([linux/memcontrol.h](https://github.com/torvalds/linux/blob/ac9dc67b730f3a1d10c5abbf91ed773d1e277646/include/linux/memcontrol.h) and [mm/memcontrol.c](https://github.com/torvalds/linux/blob/ac9dc67b730f3a1d10c5abbf91ed773d1e277646/mm/memcontrol.c)) are public in the kernel. I hope to continue this discussion on [LKML](https://lkml.org/) soon.

Others have tried [similar things modifying the proc filesystem directly](https://lkml.org/lkml/2012/5/28/299), but that is unlikely to be merged to the mainstream kernel if it affects all users of the proc filesystem. It would either need to be a custom filesystem (like [procg](https://github.com/fabiokung/procg/)) or a custom mount option to proc. E.g.:

|  |
| --- |
| mount -t proc -o meminfo-from-cgroup none /path/to/container/proc |

**FUSE**

There is also a group of kernel developers advocating that this would be better served by something outside of the kernel, in userspace, making /proc/meminfo[be a virtual file](https://lkml.org/lkml/2012/5/29/75) that collects information elsewhere and formats it appropriately.

[FUSE](http://fuse.sourceforge.net/) can be used to implement a filesystem in userspace to do just that. [Libvirt went down that path](http://www.redhat.com/archives/libvir-list/2012-November/msg00540.html) with its [libvirt-lxc driver](http://libvirt.org/drvlxc.html#fsmounts). There were attempts to integrate a FUSE version of /proc/meminfo into [LXC](https://lists.linuxcontainers.org/pipermail/lxc-devel/2009-September/000434.html) [too](https://github.com/hallyn/procfs).

Even though there is a very nice implementation of [FUSE in pure Go](http://bazil.org/fuse/), and that I am excited with the idea to contribute a plugin/patch to Docker using it, at Heroku we (myself included) have a lot of resistance against using FUSE in production.

This is mainly due to bad past experiences with FUSE filesystems (sshfs, s3fs) and the increased [surface area for attacks](http://www.cvedetails.com/vulnerability-list/vendor_id-3090/Fuse.html). My research so far has revealed that the situation may be much better nowadays, and I would even be willing to give it a try if there were not [other problems with using fuse](https://lists.linuxcontainers.org/pipermail/lxc-users/2010-May/000348.html) to replace the proc filesystem.

I am also not comfortable with making my containers dependent on an userspace daemon that serves FUSE requests. What happens when that daemon crashes? All containers in the box are probably left without access to their /proc/meminfo. Either that, or having to run a different daemon per container. Hundreds of containers in a box would require hundreds of such daemons. Ugh.

**/proc is not the only issue:**sysinfo

Even if we could find a solution to containerize /proc/meminfo with which everyone is happy, it **would not be enough**.

Linux also provides the [sysinfo(2)](http://man7.org/linux/man-pages/man2/sysinfo.2.html) syscall, which returns information about system resources (e.g. memory). As with /proc/meminfo, it is not containerized: it always returns metrics for the box as a whole.

I was surprised while [testing my proc replacement](https://github.com/fabiokung/kdevbox) ([procg](https://github.com/fabiokung/procg/)) that it did not work with [Busybox](http://www.busybox.net/). Later, I discovered that [the Busybox’s implementation of free](http://git.busybox.net/busybox/tree/procps/free.c)does not use /proc/meminfo. Guess what? It uses sysinfo(2). What else out there could also be using sysinfo(2) and be broken inside containers?

**ulimit, setrlimit**

On top of cgroup limits, Linux processes are also subject to resource limits applied to them individually, via [setrlimit(2)](http://linux.die.net/man/2/setrlimit).

Both cgroup limits and rlimit apply when memory is being allocated by a process.

**systemd**

Soon, [cgroups are going to be managed by systemd](http://www.freedesktop.org/wiki/Software/systemd/ControlGroupInterface/). All operations on cgroups are going to be done through API calls to systemd, over DBUS (or a shared library).

That makes me think that systemd could also expose a consistent API for processes to query their available memory.

But until then…

**Solution?**

Some kernel developers (and I am starting to agree with them) believe that the best option is an userspace library that processes can use to query their memory usage and available memory.

libmymem would do all the hard work of figuring out where to pull numbers from (/proc vs. cgroup vs. getrlimit(2) vs. systemd, etc.). I am considering starting one. New code could easily benefit from it, but it is unlikely that all existing tools (free, top, etc.) will just switch to it. For now, we might need to encourage people to stop using those tools inside containers.

I hope my unfortunate realization – figuring out how much memory you can use inside a container is harder than it should be – helps people better understand the problem. Please leave a comment below and let me know what you think.

1. To be fair, I don’t really know what NewRelic is doing these days, I am just using them as an example. They may be reading memory metrics in a different way (maybe aggregating information from proc/\*/smaps). Pardon my ignorance.