Resource Management: Beancounters

Pavel Emelianov xemul@openvz.org
Denis Lunev den@openvz.org
Kirill Korotaev dev@openvz.org

July 31, 2007

Abstract

The paper outlines various means of resource management available in the Linux kernel, such as per-process limits (the setrlimit(2) interface), shows their shortcomings, and illustrares the need for another resource control mechanism: beancounters.

Beancounters are a set of per-process group parameters (proposed and implemented by Alan Cox and Andrey Savochkin and further developed for OpenVZ) which can be used with or without containers.

Beancounters history, architecture, goals, efficiency, and some in-depth implementation details are given.

1 Current state of resource management in the Linux kernel

Currently the Linux kernel has only one way to control resource consumption of running processes – it is UNIX-like resource limits (rlimits).

Rlimits set upper bounds for some resource usage parameters. Most of these limits apply to a single process, except for the limit for the number of processes, which applies to a user.

The main goal of these limits is to protect processes from an **accidental** misbehavior (like infinite memory consumption) of other processes. A better way to organize such a protection would be to set up a **minimal** amount of resources that are **always** available to the process. But the old way (specifying upper limits to catch some cases of excessive consumption) may work, too.

It is clear that the reason for introducing limits was the protection against an accidental misbehavior of processes. For example, there are separate limits for the data and stack size, but the limit on the total memory consumed by the stack, data, BSS, and memory mapped regions does not exist. Also, it is worth to note that the `RLIMIT_CORE` and `RLIMIT_RSS` limits are mostly ignored by the Linux kernel.

Again, most of these resource limits apply to a single process, which means, for example, that all the memory may be consumed by a single user running the appropriate number of processes. Setting the limits in such a way as to have the value of multiplying the per-process limit by the number of processes staying below the available values is impractical.

2 Beancounters

For some time now Linus has been accepting patches adding the so-called namespaces into the kernel. Namespaces allow tasks to observe their own set of particular kernel objects such as IPC objects, PIDs, network devices and interfaces, etc. Since the kernel provides such a grouping for tasks, it is necessary to control the resource consumption of these groups.

The current mainline kernel lacks the ability of tracking the kernel resource usage in terms of arbitrary task groups and this ability is required rather badly.

The list of the resources that the kernel provides is huge but the main resources are:

- Memory,
- CPU,

- IO bandwidth,
- Disk space,
- Networking bandwidth.

This article describes the architecture the OpenVZ team proposes for controlling the first resource (memory) called "beancounters". Other resource control mechanisms are out of the scope of this article.

2.1 Beancounters history

All the deficiencies of the per-process resource accounting were noticed by Alan Cox and Andrey Savochkin long ago. Then Alan introduced an idea that crucial kernel resources must be handled in terms of groups and set the "beancounter" name for this group. He also stated that once a task moves to a separate group it never comes back and neither do the resources allocated on its requests.

These ideas were further developed by Andrey Savochkin, who proposed the first implementation of beancounters [UB patch]. It included the tracking of process page tables, the numbers of tasks within a beancounter, and the total length of mappings. Further versions included such resources as file locks, pseudo terminals, open files, etc.

Nowadays the beancounters used by OpenVZ control the following resources:

- kernel memory,
- user-space memory,
- number of tasks,
- number of open files and sockets,
- number of PTYs, file locks, pending signals and iptable rules,
- total size of network buffers,
- active dentry cache size, i.e. the size of the dentries that cannot be shrunk,
- dirty page cache that is used to track the IO activity.

2.2 The beancounters basics

The core object is the beancounter itself. The beancounter represents a task group which is a resource consumer.

Basically a beancounter consists of an ID to make it possible to address the beancounter from the userspace for changing its parameters and the set of usage-limit pairs to control the usage of several kernel resources.

More precisely each beancounter contains not just usage-limit pairs but a more complicated set. It includes the usage, limit, barrier, fail counter and maximal usage values.

The notion of "barrier" has been introduced to make it possible to start rejecting the allocation of a resources before the limit has been hit. For example when a beancounter hits the mappings barrier, the subsequent `sbrk` and `mmap` calls start to fail, though `execve`, which maps some areas, still works. This allows the administrator to "warn" the tasks that a beancounter is short of resources before the corresponding group hits the limit and the tasks start dying due to unrecoverable rejections of resource allocation.

Allocating a new resource is preceded with its "charging" to the beancounter the current task belongs to. Essentially the charging consists of an atomic checking that the amount of resource consumed doesn't exceed the limit, and adding the resource size to the current usage.

Here is a list of memory resources controlled by the beancounters subsystem:

- total size of allocated kernel space objects,
- size of network buffers,
- lengths of mappings,
- number of physical pages.

Below are some details on the implementation.

3 Memory management with beancounters

This section describes the way beancounters are used to track memory usage. The kernel memory is accounted separately from the userspace one. First, the userspace memory management is described.

3.1 Userspace memory management

The kernel defines two kinds of requests related to memory allocation.

1. The request to allocate a memory region for physical pages. This is done via the `mmap(2)` system call. The process may only work within a set of mmap-ed regions, which are called VMAs (virtual memory areas). Such a request does not lead to allocation of any physical memory to the task. On the other hand, it allows for a *graceful reject* from the kernel space, i.e. an error returned by the system call makes it possible for the task to take some actions rather than die suddenly and silently;

2. The request to allocate a physical page within one of the VMAs created before. This request may also create some kernel space objects, e.g. page tables entries. The only way to reject this request is to send a signal – `SEGV`, `BUS`, or `KILL` – depending on the failure severity. This is not a good way of rejecting as not every application expects critical signals to come during normal operation.

The beancounters introduce the "unused page" term to indicate a page in the VMA that has not yet been touched, i.e. a page whose VMA has already been allocated with `mmap`, but the page itself is not yet there. The "used" pages are physical pages.

Kernel VMAs may be classified as:

- reclaimable VMAs, which are backed by some file on the disk. I.e. when the kernel needs some more memory, it can safely push the pages from this VMA to the disk with almost no limitation;

- unreclaimable VMAs, which are not backed by any file and the only way to free the pages within such VMAs is push the page to the swap space. This way has certain limitations in that the system may have no swap at all or the swap size may be too small. The reclamation of pages from this kind of VMAs is more likely to fail in comparison with the previous kind.

In the terms defined above, the OpenVZ beancounters account for the following resources:

- the number of used pages within all the VMAs,

- the sum of unused and used pages within unreclaimable VMAs and used pages in reclaimable VMAs.

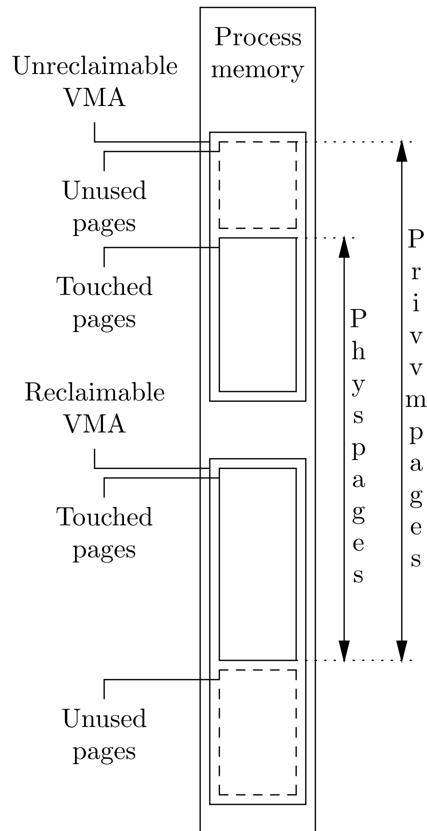

Figure 1: User space memory management

The first resource is account-only (i.e. not limited) and is called "physpages". The second one is limited in respect of only unused pages allocation and is called "privvmpages". This is illustrated on Fig. 1.

The only parameter that remains unlimited – the size of pages touched from a disk file – does not create a security hole since the size of files is limited by the OpenVZ disk quotas.

3.2 Physpages management

While privvmpages accounting works with whole pages, physpages accounts for pages fractions in the case some pages are shared among beancounters [RSS]. This may happen if a task changes its

beancounter or if tasks from different groups map and use the same file.

This is how it works. There is a many-to-many dependence between the mapped pages and the beancounters in the system. This dependence is tracked by a ring of *page beancounters* associated with each mapped page (see Fig. 2). What is important is that each page has its associated circular list of page beancounters and the head of this list has a special meaning.

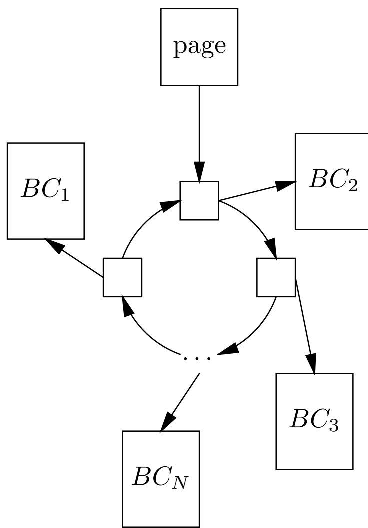

Figure 2: Page shared among beancounters

Naturally, if a page is shared among N beancounters, each beancounter is to get a $\frac{1}{N}$-th part of it. This approach would be the most precise, but also the worst, because adding a page to a new beancounter would require an update of all the beancounters among which the page is already shared.

Instead of doing that, parts of the page equal to

$$\frac{1}{2^{shift(page,beancounter)}}$$

are charged, where $shift(page, beancounter)$ is calculated for

$$\sum_{beancounters} \frac{1}{2^{shift(page,beancounter)}} = 1$$

to be true for each page.

When mapping a page into a new beancounter, half of the part charged to the head of the page's beancounters list is moved to the new beancounter. Thus when the page is sequentially mapped to 4 different beancounters, its fractions would look like

	bc_1	bc_2	bc_3	bc_4
1	1			
2	$\frac{1}{2}$	$\frac{1}{2}$		
3	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{4}$	
4	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$

When unmapping a page from a beancounter, the fraction of the page charged to this beancounter is returned to one or two of the beancounters on the page list.

For example, when unmapping the page from bc_4 the fractions would change like

$$\{\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}\} \rightarrow \{\frac{1}{2}, \frac{1}{4}, \frac{1}{4}\}$$

i.e. a fraction from bc_4 was added to only one beancounter – bc_3.

Next, when unmapping the page from bc_3 with fraction of $\frac{1}{2}$ its charge will be added to two beancounters to keep fractions be the powers of two:

$$\{\frac{1}{2}, \frac{1}{4}, \frac{1}{4}\} \rightarrow \{\frac{1}{2}, \frac{1}{2}\}$$

With that approach the following has been achieved:

- algorithm of adding and removing references to beancounters has $O(1)$ complexity;
- the sum of the physpages values from all the beancounters is the number of RAM pages actually used in the kernel.

3.2.1 Dirty page cache management

The above architecture is used for dirty page cache and thus IO accounting.

Let's see how IO works in the Linux kernel [RLove]. Each page may be either *clean* or *dirty*. Dirty pages are marked with the bit of the same name and are about to be written to disk. The main point here is that dirtying a page doesn't imply that it will be written to disk immediately. When the actual writing starts, the task (and thus the beancounter) that made the page dirty may already be dead.

Another peculiarity of IO is that a page may be marked as dirty in context different from the one

that really owns the page. Arbitrary pages are unmapped when the kernel shrinks the page cache to free more memory. This is done by checking the `pte` dirty flag set by a CPU.

Thus it is needed to save the context a page became dirty in till the moment the page becomes clean, i.e. its data is written to disk. To track the context in question, an IO beancounter is added between the page and its page beancounters ring (see Fig. 3). This IO beancounter keeps the appropriate beancounter while the page stays dirty.

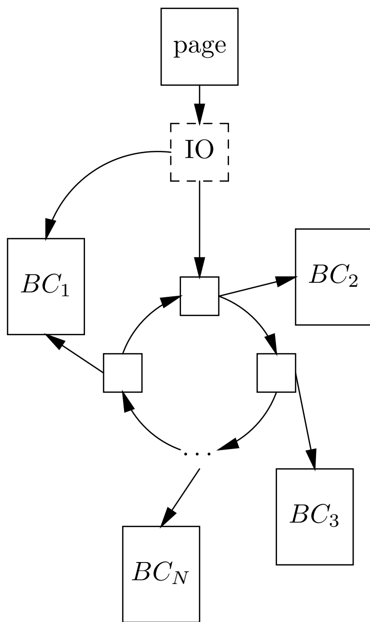

Figure 3: Tracking the dirtier of the page

To determine the context of a page, the page beancounter ring mentioned above is used. When a page becomes dirty in a synchronous context, e.g. a regular `write` happens, the current beancounter is used. When a page becomes dirty in an arbitrary context, e.g. from a page shrinker, the first beancounter from this ring is used.

3.3 Kernel memory management

Privvmpages allow a user to track the memory usage of tasks and give the applications a chance for graceful exit rather than killing, but do not provide any protection. Kernel memory size – the "kmemsize" in the beancounters terms – is the way to protect the system from a DoS attack from userspace.

The kernel memory is a scarce resource on 32-bit systems due to the limited normal zone size. But even on 64-bit systems applications causing the kernel to consume the unlimited amount of RAM can easily DoS it.

There is a great difference between the user memory and the kernel memory which results in dramatical difference in implementation.

When a user page is freed, the task and thus the beancounter it belongs to is always well known. When a kernel space object is freed, the beancounter it belongs to is almost never known due to refcounting mechanism and RCU. Thus each kernel object should carry a pointer on the beancounter it was allocated by.

The way beancounters store this pointer differs between different techniques of object allocating.

There are three mechanisms for object allocation in the kernel:

1. *buddy page allocator* – the core mechanism used directly and by the other two. To track the beancounter of the directly allocated page there is an additional pointer on `struct page`. Another possible way would be to allocate a mirrored to `mem_map` array of pointers, or reuse `mapping` field on `struct page`;

2. *vmalloc*. To track the owner of vmalloc-ed object the owner of the first (more precisely – the zeroth) page is used. This is simple as well;

3. *kmalloc* – the way to allocate small objects (down to 32 bytes of size). Many objects allocated with kmalloc must be tracked and thus have a pointer to the corresponding beancounter. First versions of beancounters changed such objects explicitly by adding `struct beancounter *owner` field in structures. To unify this process in new versions, `mm/slab.c` is modified by adding an array of pointers on beancounters at the tail of each slab (see Fig. 4). This array is indexed with the sequence number of the object on the slab.

Such an architecture has two potential side effects:

1. slabs will become shorter, i.e. one slab will carry less objects than it did before;

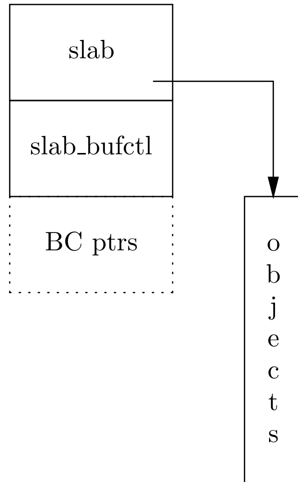

Figure 4: Extending a slab for the kernel memory tracking

2. slabs may become "offslab" as the gap will exceed the offslab border.

Slab size	# of objects		offslab-ness	
	before	after	before	after
size-32	113	101	−	−
size-64	59	56	−	−
size-128	30	29	−	−
size-256	15	15	−	−
size-512	8	8	+	+
size-1024	4	4	+	+
size-2048	2	2	+	+
size-4096...	1	1	+	+

Table 1: The comparison of size-X caches with and without beancounters patch

Table 1 shows the results for `size-X` caches. As seen from this table less than 1% of available objects from "on-page" slabs is lacked. "Offslab" caches do not lack any objects and no slabs become offslab.

3.4 Network buffers management

Network buffers management [AFTM] fundamentally differs for the send and receive buffers. Basically, the host does not control the incoming traffic and fully control the outgoing traffic.

Unfortunately, the mainstream kernel socket accounting mechanism cannot be reused, as it has known deficiencies:

- slab overhead is not included into the accounted packet size. For Ethernet the difference is around 25%-30% as `size-2048` slab is used for packets. Unfortunately, `skb->truesize` calculations can't be changed without massive TCP/IP stack fix, as this would lead to serious performance degradation due to TCP window reduction;

- the accounting is not strict, and limits can be over-used.

3.4.1 Send buffer space

TCPv4 now and TCPv6 with DCCP in the future are accounted separately from all the other outgoing traffic. Only the packets residing in the socket send queue are charged since skb "clones" sent to device for transmission have a very limited lifetime.

Netlink send operations charging is not uniform in respect to send direction. The buffers are charged for user socket only even if they are sent from the kernel. This is fair as, usually, data stream from the kernel to a user is initiated by the user.

Beancounters track memory usage with the appropriate overheads on per-socket basis. Each socket has a guarantee calculated as

$$\frac{limit - barrier}{N_{sockets}},$$

where $N_{sockets}$ is the limit of sockets of appropriate type on the beancounter.

Beancounters don't break a semantics of the `select` system call, i.e. if it returns that the socket can be written to, `write` will send at least one byte. To achieve this goal an additional field for the socket structure had been introduced, namely, `poll_reserve`. So, actual resource usage is shown as

$$\sum_{s \in sockets} (s_{poll_reserv} + \sum_{skb \in s_{write_queue}} skb_{size})$$

3.4.2 Receive buffer space

Management of the receive buffers for non-TCP sockets is simple. Incoming packets are simply

dropped if the limit is hit. This is normal for raw IP sockets and UDP, as there is no protocol guarantee that the packet will be delivered.

Though, there is a problem with the netlink sockets. In general, the user (`ip` or similar tool) sends a request to the kernel and starts waiting for an answer. The answer may never arrive as the kernel does not retransmit netlink responses. Though, the practical impact of this deficiency is acceptable.

TCP buffer space accounting is mostly the same except for the guarantee for one packet. The amount equal to $N_{sockets} * max_advmss$ is reserved for this purpose. Beancounters rely on the generic code for TCP window space shrinking. Though, a better window management policy for all sockets inside a beancounter is being investigated.

4 Performance

The OpenVZ team spent a lot of time improving the performance of the beancounters patches. The following techniques are employed:

- **Pre-charging of resources on a task creation.** When the task later tries to charge a new portion of resource, it may take the desired amount from this reserve. The same technique is used in network buffers accounting, but pre-charge is stored on socket;

- **On-demand accounting.** I.e. per-group accounting is not performed when the overall resource consumption is low enough. When a system becomes low of some resource per-beancounter accounting is turned on and vice-versa – when a system has a lot of free resources, per-beancounter accounting is turned off. Nevertheless this switch is rather slow as it implies seeking for all resources belonging to a beancounter and thus it should happen rarely.

The results of unixbench test are shown for the following kernels:

- Vanilla 2.6.18 kernel,

- OpenVZ 2.6.18-028stab025 kernel,

- OpenVZ 2.6.18-028stab025 kernel without pages sharing accounting.

Test name	Vanilla	OVZ	%
Process Creation	7456	7288	97%
Execl Throughput	2505	2490	99%
Pipe Throughput	4071	4084	100%
Shell Scripts	4521	4369	96%
File Read	1051	1041	99%
File Write	1070	1086	101%

Table 2: Unixbench results comparison (p.1)

Test name	Vanilla	OVZ	%
Process Creation	7456	6788	91%
Execl Throughput	2505	2290	91%
Pipe Throughput	4071	4064	99%
Shell Scripts	4521	3969	87%
File Read	1051	1031	98%
File Write	1070	1066	99%

Table 3: Unixbench results comparison (p.2)

Tests were run on Dual-Core Intel® Xeon™ CPU 3.20GHz machine with 3Gb of RAM.

Table 2 shows the results of comparison of the vanilla kernel vs the OpenVZ kernel without pages sharing accounting and table 3 – for vanilla kernel vs full beancounters patch.

5 Conclusions

Described in this article is the memory management made in "beancounters" subsystem. The userspace memory including RSS accounting and the kernel memory including network buffers management were described.

Beancounters architecture has proven its efficiency and flexibility. It has been used in OpenVZ kernels for all the time OpenVZ project exists.

The questions that are still unsolved are:

- TCP window management based on the beancounter resource availability;

- on-demand RSS accounting;

- integration with the mainstream kernel.

References

[UB patch] Andrey Savochkin, *User Beancounter Patch*,

http://mirrors.paul.sladen.org/ftp.sw.com.sg/pub/Linux/people/saw/kernel/user_beancounter/UserBeancounter.html

[RSS] Pavel Emelianov, *RSS fractions accounting* http://wiki.openvz.org/RSS_fractions_accounting

[AFTM] ATM Forum, *Traffic Management Specification, version 4.1*

[RLove] Robert Love, *Linux Kernel Development (2nd Edition)*, ch. 15. Novell Press, January 12, 2005.