

# mm/ hacks -- Generic memory store?

---

*Source:* <http://linux.derkeiler.com/Mailing-Lists/Kernel/2006-01/msg00300.html>

---

- *From:* John Richard Moser <[nigelenki@xxxxxxxxxxxx](mailto:nigelenki@xxxxxxxxxxxx)>
  - *Date:* Mon, 02 Jan 2006 23:36:15 -0500
- 

-----BEGIN PGP SIGNED MESSAGE-----

Hash: SHA1

I want to do a memory management hack for generic memory stores. Basically, I want to abstract swap from the kernel; instead, I want a "store" for excess memory before OOM. I'm wondering currently the usefulness and feasibility of this idea.

Consider the kernel's current memory management:

RAM <-> swap

Whereas the new management scheme would be only slightly different:

RAM <-> store

Now, the memory store infrastructure would have its own way of handling memory. This would be abstracted from the kernel.

store <-> swap

Thus, the kernel would operate in typical fashion:

RAM <-> store <-> swap

By giving some level of extensibility to the memory store infrastructure, however, more complex pipelines can be created.

RAM <-> store <-> compress <-> swap

The memory store could be used both by disk cache and by process memory management. This would be best facilitated by allowing stored pages to be marked as persistent or non-persistent. Disk cache would of course be non-persistent as long as it could be re-read from disk; other data would be persistent.

With a persistent/non-persistent mark, the memory store could then free non-persistent data at its discretion, notifying a "destructor" function pointer of the change so that associated retrieval data can be freed;

the alternative would be to lazy destruct, which means disk cache would only find out some cache data was gone when that part of a file is re-accessed. Unfortunately, this lazy approach would leave files that are cached and never revisited with a handle in the disk cache subsystem forever, unless you want to periodically garbage collect.

The handle in the memory store to the page would probably be handled in the same way as it is for swap currently. In any case, the handle has to uniquely identify the exact page being stored. This handle could be used by the data store's swap module when writing the data to disk as well.

Suspend-to-Disk would have to be rewritten to work through the data store, as swap would now be abstracted from the kernel. This requires two things of the data store. First, it has to be aware of off-RAM persistent storage options, such as swap. Second, it has to be able to take a signal that means "flush to persistent storage."

Depending on implementation, it may be required to have the data store be told to store certain things certain ways for Suspend-to-Disk to work. For example, the swap file would have to be given the attributes of holding a suspended session for a specific kernel version. Reading this data would happen through the data store, possibly via signalling from an initrd script.

The advantages to this approach are that storage in tight memory situations can be altered easily. For example, a plug-in for the data store would be capable of providing compression or encryption. These plug-ins would be given priorities (compression first, encryption second) and attributed as transient/persistent and terminal/transparent.

A transient data store is one that stores data in memory. When the power goes, so does the data. Persistent data stores, on the other hand, store data in ways that can handle a power cycle. Typically a transient store should be transparent, and a persistent store should be terminal; but this may not always be true.

A terminal store stores data in a final state. Data in a terminal store will be pushed no further down the chain; it has to be read back out of that store before put into a new one. On the other hand, a transparent store would host data such that it could be directly processed into another store. Transparent stores should be in-memory so that they can be operated on as memory.

Basically, a transparent store would allocate pages of memory through the memory store API. This would simply allocate memory through the real memory management APIs, marking it as being non-swappable (or rather non-storable). The transparent stores would then do their thing on the pages given them, store them in allocated memory (locking the pages they're accessing), and be done with it. The memory store would operate directly on these pages when passing them to the next level.

## mm/ hacks -- Generic memory store?

The memory store would have to implement its own prioritising; of course the prioritising is basically FIFO here anyway. A page enters the store when it's put there, and leaves it when it's requested; it doesn't get read at will and still persist. Thus, the store would assume that the kernel is doing all the prioritising and telling the memory store in order what data it thinks has the least value.

When memory gets tight, the store pushes the most aged stored data to the next level. So for example the oldest data may be compressed, then swapped out (or freed, in the case of disk cache). A more interesting scheme WRT compression would be to have multiple levels of compression, where data is stored; compressed (terminal); compressed more intensively (terminal); compressed even more intensively (transparent); encrypted (transparent), and then swapped (terminal). This gives the following growth model:

```
RAM <-> store <-> Compress 1 (TERM)
4k 4k | 3k
-> Compress 2 (TERM)
| 2k
-> Compress 3 <-
1k |
-> Encrypt <-> Swap
1k 0k (swapped out)
```

NOTE: The chunks that are less than 4k are managed with other chunks in the stores. In the Compress 3 and Encrypt transparent data stores, these 1k chunks are passed as part of a full 4k page that includes other stored pieces of data.

Here we can see that Compress 1 and 2 are terminal, and get de-compressed and fed into each in turn, finally to Compress 3, then Encrypt, then Swap. We can also see that it may be wise to mark some of these as destructor points for transient data; for example, disk cache may get freed at Encrypt.

Overall, implementing this would probably be rather tough. The most difficult design issue would likely be suspend-to-disk through this using swap. The solution to that particular issue should be abstracted through the data store; this would allow for storage methods like network storage (swap to an encrypted swap network protocol or NFS), although psychotic in nature, to be used for such things.

So the questions that arise are:

1. Is this useful?

I think it would be useful to be able to add abstraction to kernel memory storage at low memory levels, to allow easy implementation of compression and encryption pre-swapping. This would even allow for modules that compress disk cache, or anything that gets swapped out

typically.

2. Is this worth implementing?

I'm unsure of the amount of work that would go into implementing this. It'd involve tearing the swap handling stuff out of the kernel and rewriting in its place this thing, then placing the swap code in as a module. More importantly, suspending to swap would be a large consideration.

3. What usefulness is of what's given as examples here?

Encrypt before swap is a nice idea, and typically in small amounts it should be practical on most systems. That isn't to say it would be friendly to microbenchmarks -- swapping would become a good deal more expensive.

Compression is of course an all win. The multi-stage compression shown here would use CPU muscle power to avoid the pains of swap; but would take a conservative approach and try to avoid using excess CPU for heavy compression unless needed. As all but the last level of compression are terminal, it would be possible also to skip to Compress 3 in periods of rapidly growing memory usage to avoid faulting through Compress 2 and Compress 1 and using 3 times as much CPU. In any case the result is better utilization of RAM; even systems with ample RAM would benefit from the increase in available disk cache.

4. What other uses are possible?

What other things do you want to do to RAM when you're low? Is there any other use? If not, is this still the cleanest route to take for the uses that are there?

-- --

All content of all messages exchanged herein are left in the Public Domain, unless otherwise explicitly stated.

Creative brains are a valuable, limited resource. They shouldn't be wasted on re-inventing the wheel when there are so many fascinating new problems waiting out there.

-- Eric Steven Raymond

-----BEGIN PGP SIGNATURE-----

Version: GnuPG v1.4.2 (GNU/Linux)

Comment: Using GnuPG with Thunderbird - <http://enigmail.mozdev.org>

iD8DBQFDuf8+hDd4aOud5P8RAiFFAJ9g9ObyyfcD7jXvmMC4nhzGR6RhogCeKKwl  
DzjoBsiyIZB93TSEcm3wViY=  
=J7Rr

-----END PGP SIGNATURE-----

--

To unsubscribe from this list: send the line "unsubscribe linux-kernel" in the body of a message to [majordomo@xxxxxxxxxxxxxxxx](mailto:majordomo@xxxxxxxxxxxxxxxx)

More majordomo info at <http://vger.kernel.org/majordomo-info.html>

Please read the FAQ at <http://www.tux.org/lkml/>

---

- Prev by Date: ***Re: [patch 00/2] improve .text size on gcc 4.0 and newer compilers***
- Next by Date: ***Inclusion of x86 64 memorize ioapic at bootup patch***
- Previous by thread: ***Re: [PATCH 0/3] msi abstractions and support for altix***
- Next by thread: ***Inclusion of x86 64 memorize ioapic at bootup patch***
- Index(es):
  - ◆ ***Date***
  - ◆ ***Thread***