Short users guide for SLUB

The basic philosophy of SLUB is very different from SLAB. SLAB
requires rebuilding the kernel to activate debug options for all
slab caches. SLUB always includes full debugging but it is off by default.
SLUB can enable debugging only for selected slabs in order to avoid
an impact on overall system performance which may make a bug more
difficult to find.

In order to switch debugging on one can add a option "slub_debug"
to the kernel command line. That will enable full debugging for
all slabs.

Typically one would then use the "slabinfo" command to get statistical
data and perform operation on the slabs. By default slabinfo only lists
slabs that have data in them. See "slabinfo -h" for more options when
running the command. slabinfo can be compiled with

gcc -o slabinfo Documentation/vm/slabinfo.c

Some of the modes of operation of slabinfo require that slub debugging
be enabled on the command line. F.e. no tracking information will be
available without debugging on and validation can only partially
be performed if debugging was not switched on.

Some more sophisticated uses of slub_debug:

Parameters may be given to slub_debug. If none is specified then full
debugging is enabled. Format:

slub_debug=<Debug-Options>   Enable options for all slabs
slub_debug=<Debug-Options>,<slab name>
                                 Enable options only for select slabs

Possible debug options are

F	Sanity checks on (enables SLAB_DEBUG_FREE. Sorry SLAB legacy issues)
Z	Red zoning
P	Poisoning (object and padding)
U	User tracking (free and alloc)
T	Trace (please only use on single slabs)
A	Toggle failslab filter mark for the cache
O	Switch debugging off for caches that would have caused higher minimum slab orders
-	Switch all debugging off (useful if the kernel is configured with CONFIG_SLUB_DEBUG_ON)

F.e. in order to boot just with sanity checks and red zoning one would specify:

```
slub_debug=FZ
```

Trying to find an issue in the dentry cache? Try

```
slub_debug=,dentry
```

to only enable debugging on the dentry cache.

Red zoning and tracking may realign the slab. We can just apply sanity checks to the dentry cache with

        slub_debug=F,dentry

Debugging options may require the minimum possible slab order to increase as a result of storing the metadata (for example, caches with PAGE_SIZE object sizes). This has a higher liklihood of resulting in slab allocation errors in low memory situations or if there's high fragmentation of memory. To switch off debugging for such caches by default, use

        slub_debug=O

In case you forgot to enable debugging on the kernel command line: It is possible to enable debugging manually when the kernel is up. Look at the contents of:

/sys/kernel/slab/<slab name>/

Look at the writable files. Writing 1 to them will enable the corresponding debug option. All options can be set on a slab that does not contain objects. If the slab already contains objects then sanity checks and tracing may only be enabled. The other options may cause the realignment of objects.

Careful with tracing: It may spew out lots of information and never stop if used on the wrong slab.

Slab merging

If no debug options are specified then SLUB may merge similar slabs together in order to reduce overhead and increase cache hotness of objects. slabinfo -a displays which slabs were merged together.

Slab validation

SLUB can validate all object if the kernel was booted with slub_debug. In order to do so you must have the slabinfo tool. Then you can do

slabinfo -v

which will test all objects. Output will be generated to the syslog.

This also works in a more limited way if boot was without slab debug. In that case slabinfo -v simply tests all reachable objects. Usually these are in the cpu slabs and the partial slabs. Full slabs are not tracked by SLUB in a non debug situation.

Getting more performance

To some degree SLUB's performance is limited by the need to take the list_lock once in a while to deal with partial slabs. That overhead is governed by the order of the allocation for each slab. The allocations can be influenced by kernel parameters:

```
slub_min_objects=x              (default 4)
slub_min_order=x                (default 0)
slub_max_order=x                (default 1)
```

slub_min_objects allows to specify how many objects must at least fit into one slab in order for the allocation order to be acceptable. In general slub will be able to perform this number of allocations on a slab without consulting centralized resources (list_lock) where contention may occur.

slub_min_order specifies a minim order of slabs. A similar effect like slub_min_objects.

slub_max_order specified the order at which slub_min_objects should no longer be checked. This is useful to avoid SLUB trying to generate super large order pages to fit slub_min_objects of a slab cache with large object sizes into one high order page.

SLUB Debug output

Here is a sample of slub debug output:

```
====================================================================
BUG kmalloc-8: Redzone overwritten
--------------------------------------------------------------------

INFO: 0xc90f6d28-0xc90f6d2b. First byte 0x00 instead of 0xcc
INFO: Slab 0xc528c530 flags=0x400000c3 inuse=61 fp=0xc90f6d58
INFO: Object 0xc90f6d20 @offset=3360 fp=0xc90f6d58
INFO: Allocated in get_modalias+0x61/0xf5 age=53 cpu=1 pid=554

Bytes b4 0xc90f6d10:  00 00 00 00 00 00 00 00 5a 5a 5a 5a 5a 5a 5a 5a
........ZZZZZZZZ
  Object 0xc90f6d20:  31 30 31 39 2e 30 30 35                         1019.005
 Redzone 0xc90f6d28:  00 cc cc cc                                     .
 Padding 0xc90f6d50:  5a 5a 5a 5a 5a 5a 5a 5a                         ZZZZZZZZ

  [<c010523d>] dump_trace+0x63/0x1eb
  [<c01053df>] show_trace_log_lvl+0x1a/0x2f
  [<c010601d>] show_trace+0x12/0x14
  [<c0106035>] dump_stack+0x16/0x18
  [<c017e0fa>] object_err+0x143/0x14b
  [<c017e2cc>] check_object+0x66/0x234
  [<c017eb43>] __slab_free+0x239/0x384
  [<c017f446>] kfree+0xa6/0xc6
  [<c02e2335>] get_modalias+0xb9/0xf5
  [<c02e23b7>] dmi_dev_uevent+0x27/0x3c
  [<c027866a>] dev_uevent+0x1ad/0x1da
  [<c0205024>] kobject_uevent_env+0x20a/0x45b
  [<c020527f>] kobject_uevent+0xa/0xf
```

```
  [<c02779f1>] store_uevent+0x4f/0x58
  [<c027758e>] dev_attr_store+0x29/0x2f
  [<c01bec4f>] sysfs_write_file+0x16e/0x19c
  [<c0183ba7>] vfs_write+0xd1/0x15a
  [<c01841d7>] sys_write+0x3d/0x72
  [<c0104112>] sysenter_past_esp+0x5f/0x99
  [<b7f7b410>] 0xb7f7b410
  =======================
```

FIX kmalloc-8: Restoring Redzone 0xc90f6d28-0xc90f6d2b=0xcc

If SLUB encounters a corrupted object (full detection requires the kernel
to be booted with slub_debug) then the following output will be dumped
into the syslog:

1. Description of the problem encountered

This will be a message in the system log starting with

```
==============================================
BUG <slab cache affected>: <What went wrong>
----------------------------------------------

INFO: <corruption start>-<corruption_end> <more info>
INFO: Slab <address> <slab information>
INFO: Object <address> <object information>
INFO: Allocated in <kernel function> age=<jiffies since alloc> cpu=<allocated by
        cpu> pid=<pid of the process>
INFO: Freed in <kernel function> age=<jiffies since free> cpu=<freed by cpu>
         pid=<pid of the process>
```

(Object allocation / free information is only available if SLAB_STORE_USER is
set for the slab. slub_debug sets that option)

2. The object contents if an object was involved.

Various types of lines can follow the BUG SLUB line:

Bytes b4 <address> : <bytes>
        Shows a few bytes before the object where the problem was detected.
        Can be useful if the corruption does not stop with the start of the
        object.

Object <address> : <bytes>
        The bytes of the object. If the object is inactive then the bytes
        typically contain poison values. Any non-poison value shows a
        corruption by a write after free.

Redzone <address> : <bytes>
        The Redzone following the object. The Redzone is used to detect
        writes after the object. All bytes should always have the same
        value. If there is any deviation then it is due to a write after
        the object boundary.

        (Redzone information is only available if SLAB_RED_ZONE is set.
        slub_debug sets that option)

Padding <address> : <bytes>
	Unused data to fill up the space in order to get the next object
	properly aligned. In the debug case we make sure that there are
	at least 4 bytes of padding. This allows the detection of writes
	before the object.

3. A stackdump

The stackdump describes the location where the error was detected. The cause
of the corruption is may be more likely found by looking at the function that
allocated or freed the object.

4. Report on how the problem was dealt with in order to ensure the continued
operation of the system.

These are messages in the system log beginning with

FIX <slab cache affected>: <corrective action taken>

In the above sample SLUB found that the Redzone of an active object has
been overwritten. Here a string of 8 characters was written into a slab that
has the length of 8 characters. However, a 8 character string needs a
terminating 0. That zero has overwritten the first byte of the Redzone field.
After reporting the details of the issue encountered the FIX SLUB message
tells us that SLUB has restored the Redzone to its proper value and then
system operations continue.

Emergency operations:

Minimal debugging (sanity checks alone) can be enabled by booting with

	slub_debug=F

This will be generally be enough to enable the resiliency features of slub
which will keep the system running even if a bad kernel component will
keep corrupting objects. This may be important for production systems.
Performance will be impacted by the sanity checks and there will be a
continual stream of error messages to the syslog but no additional memory
will be used (unlike full debugging).

No guarantees. The kernel component still needs to be fixed. Performance
may be optimized further by locating the slab that experiences corruption
and enabling debugging only for that cache

I.e.

	slub_debug=F,dentry

If the corruption occurs by writing after the end of the object then it
may be advisable to enable a Redzone to avoid corrupting the beginning
of other objects.

	slub_debug=FZ,dentry

Christoph Lameter, May 30, 2007