

# [PATCH] SLUB The unqueued slab allocator V3

---

*Source:* <http://linux.derkeiler.com/Mailing-Lists/Kernel/2007-02/msg10178.html>

---

- *From:* Christoph Lameter <[clameter@xxxxxxxxxxxxx](mailto:clameter@xxxxxxxxxxxxx)>
  - *Date:* Wed, 28 Feb 2007 11:20:44 -0800 (PST)
- 

## V2->V3

- Debugging and diagnostic support. This is runtime enabled and not compile time enabled. Runtime debugging can be controlled via kernel boot options on an individual slab cache basis or globally.
- Slab Trace support (For individual slab caches).
- Resiliency support: If basic sanity checks are enabled (via F f.e.) (boot option) then SLUB will do the best to perform diagnostics and then continue (i.e. mark corrupted objects as used).
- Fix up numerous issues including clash of SLUBs use of page flags with i386 arch use for pmd and pgds (which are managed as slab caches, sigh).
- Dynamic per CPU array sizing.
- Explain SLUB slabcache flags

Note that SLUB will warn on zero sized allocations. SLAB just allocates some memory. So some traces from the usb subsystem etc should be expected. There are very likely also issues remaining in SLUB.

## V1->V2

- Fix up various issues. Tested on i386 UP, X86\_64 SMP, ia64 NUMA.
- Provide NUMA support by splitting partial lists per node.
- Better Slab cache merge support (now at around 50% of slabs)
- List slab cache aliases if slab caches are merged.
- Updated descriptions /proc/slabinf output

This is a new slab allocator which was motivated by the complexity of the existing code in mm/slab.c. It attempts to address a variety of concerns with the existing implementation.

### A. Management of object queues

A particular concern was the complex management of the numerous object queues in SLAB. SLUB has no such queues. Instead we dedicate a slab for each allocating CPU and use objects from a slab directly instead of queueing them up.

### B. Storage overhead of object queues

SLAB Object queues exist per node, per CPU. The alien cache queue even

## [PATCH] SLUB The unqueued slab allocator V3

has a queue array that contain a queue for each processor on each node. For very large systems the number of queues and the number of objects that may be caught in those queues grows exponentially. On our systems with 1k nodes / processors we have several gigabytes just tied up for storing references to objects for those queues This does not include the objects that could be on those queues. One fears that the whole memory of the machine could one day be consumed by those queues.

### C. SLAB meta data overhead

SLAB has overhead at the beginning of each slab. This means that data cannot be naturally aligned at the beginning of a slab block. SLUB keeps all meta data in the corresponding page\_struct. Objects can be naturally aligned in the slab. F.e. a 128 byte object will be aligned at 128 byte boundaries and can fit tightly into a 4k page with no bytes left over. SLAB cannot do this.

### D. SLAB has a complex cache reaper

SLUB does not need a cache reaper for UP systems. On SMP systems the per CPU slab may be pushed back into partial list but that operation is simple and does not require an iteration over a list of objects. SLAB expires per CPU, shared and alien object queues during cache reaping which may cause strange hold offs.

### E. SLAB has complex NUMA policy layer support

SLUB pushes NUMA policy handling into the page allocator. This means that allocation is coarser (SLUB does interleave on a page level) but that situation was also present before 2.6.13. SLABs application of policies to individual slab objects allocated in SLAB is certainly a performance concern due to the frequent references to memory policies which may lead a sequence of objects to come from one node after another. SLUB will get a slab full of objects from one node and then will switch to the next.

### F. Reduction of the size of partial slab lists

SLAB has per node partial lists. This means that over time a large number of partial slabs may accumulate on those lists. These can only be reused if allocator occur on specific nodes. SLUB has a global pool of partial slabs and will consume slabs from that pool to decrease fragmentation.

### G. Tunables

SLAB has sophisticated tuning abilities for each slab cache. One can manipulate the queue sizes in detail. However, filling the queues still requires the uses of the spin lock to check out slabs. SLUB has a global parameter (min\_slab\_order) for tuning. Increasing the minimum slab order can decrease the locking overhead. The bigger the slab order the

less motions of pages between per CPU and partial lists occur and the better SLUB will be scaling.

### G. Slab merging

We often have slab caches with similar parameters. SLUB detects those on boot up and merges them into the corresponding general caches. This leads to more effective memory use. About 50% of all caches can be eliminated through slab merging. This will also decrease slab fragmentation because partial allocated slabs can be filled up again. Slab merging can be switched off by specifying `slub_nomerge` on boot up.

Note that merging can expose heretofore unknown bugs in the kernel because corrupted objects may now be placed differently and corrupt differing neighboring objects. Enable sanity checks to find those.

### H. Diagnostics

The current slab diagnostics are difficult to use and require a recompilation of the kernel. SLUB contains debugging code that is always available (but is kept out of the hot code paths). SLUB diagnostics can be enabled via the "slab\_debug" option. Parameters can be specified to select a single or a group of slab caches for diagnostics. This means that the system is running with the usual performance and it is much more likely that race conditions can be reproduced.

### I. Resiliency

If basic sanity checks are on then SLUB is capable of detecting common error conditions and recover as best as possible to allow the system to continue.

### J. Tracing

Tracing can be enabled via the `slab_debug=T,<slabcache>` option during boot. SLUB will then protocol each action on that slabcache and dump the object contents on free.

Tested on:

i386 SMP, x86\_64 UP + SMP + NUMA emulation, IA64 NUMA + Simulator

### SLUB Boot options

`slub_nomerge` Disable merging of slabs  
`slub_min_order=x` Require a minimum order for slab caches. This increases the managed chunk size and therefore reduces meta data and locking overhead.  
`slub_debug` Enable all diagnostics for all caches  
`slub_debug=<options>` Enable selective options for all caches

## [PATCH] SLUB The unqueued slab allocator V3

slub\_debug=<o>,<cache> Enable selective options for a certain set of caches

Available Debug options

F Double Free checking, sanity and resiliency

R Red zoning

P Object / padding poisoning

U Track last free / alloc

T Trace all allocs / frees (only use on individual slabs).

To use SLUB: Apply this patch and then select SLUB as the default slab allocator. The output of /proc/slabinfo will then change. Here is a sample (this is an UP/SMP format. The NUMA display will show on which nodes the slabs were allocated). Flags are

a Cpu-cache Align requested

A Hardware Align required

C Constructor

d DMA cache

D Destructor

F Double free checking/Sanity

p Panic on failure

P Poisoning

r Objects are reclaimable

R RCU destroy

S Memory Spreading

U User Tracking

T Tracing

Z Red Zone

# name <objects> <order> <objsize> <slabs>/<partial>/<cpu> <flags>

nfs\_direct\_cache 0 0 120 0/0/0 Sr

nfs\_write\_data 36 0 768 8/0/1 a

nfs\_inode\_cache 0 0 944 0/0/0 CSr

nfsd4\_delegations 0 0 656 0/0/0

nfsd4\_stateowners 0 0 424 0/0/0

rpc\_inode\_cache 6 1 704 1/0/1 CSra

mqueue\_inode\_cache 1 1 832 1/0/1 Ca

udf\_inode\_cache 0 0 608 0/0/0 CSr

fuse\_request 0 0 584 0/0/0

fuse\_inode 0 0 576 0/0/0 Ca

isofs\_inode\_cache 0 0 584 0/0/0 CSr

reiser\_inode\_cache 1721 0 640 287/0/1 CSr

inotify\_watch\_cache 4 0 72 1/0/1 p eventpoll\_pwq/nfsd4\_files

fasync\_cache 0 0 24 0/0/0 p

shmem\_inode\_cache 1328 0 720 266/1/1 C

posix\_timers\_cache 0 0 136 0/0/0

xfrm\_dst\_cache 0 0 384 0/0/0 pa

ip\_dst\_cache 137 0 320 12/0/1 pa kioclx/rpc\_tasks

[PATCH] SLUB The unqueued slab allocator V3

UDP 49 1 704 5/1/1 a RAW/UDP-Lite/nfs\_read\_data  
TCP 18 1 1472 4/2/1 a  
scsi\_io\_context 0 0 112 0/0/0  
blkdev\_ioc 26 0 56 1/0/1 p  
blkdev\_queue 3 1 1448 1/0/1 p  
blkdev\_requests 12 0 280 2/1/1 p  
sock\_inode\_cache 71 0 640 12/0/1 CSra  
file\_lock\_cache 5 0 176 1/0/1 Cp  
Acpi-Parse 1 0 40 1/0/1 inotify\_event\_cache/dnotify\_cache  
proc\_inode\_cache 267 0 568 39/4/1 CSr  
sigqueue 0 0 160 1/0/1 p  
radix\_tree\_node 5437 0 560 777/0/1 Cp  
bdev\_cache 6 0 768 2/1/1 CSrpa  
sysfs\_dir\_cache 5935 0 80 117/0/1 Acpi-State  
inode\_cache 2691 0 536 385/2/1 CSrp  
dentry\_cache 7405 0 192 353/1/1 Srp  
idr\_layer\_cache 79 0 536 12/0/1 C  
buffer\_head 4901 0 112 137/0/1 CSrp  
mm\_struct 35 1 832 5/2/1 pa  
vm\_area\_struct 1562 0 168 66/2/1 p  
signal\_cache 120 0 640 20/0/1 pa files\_cache/UNIX  
sighand\_cache 62 2 2112 9/1/1 CRpa  
task\_struct 75 2 1728 11/9/1 p  
anon\_vma 534 0 24 4/3/1 CRp  
kmallo-192 389 0 192 19/0/1 tw\_sock\_TCP/eventpoll\_epi  
kmallo-96 1186 0 96 29/1/1  
kmallo-262144 0 6 262144 0/0/0  
kmallo-131072 0 5 131072 0/0/0  
kmallo-65536 0 4 65536 0/0/0  
kmallo-32768 0 3 32768 0/0/0  
kmallo-16384 0 2 16384 0/0/0  
kmallo-8192 15 1 8192 15/0/0  
kmallo-4096 92 0 4096 92/0/0 names\_cache/biovec-256/sgpool-128  
kmallo-2048 175 0 2048 88/1/1 biovec-128/sgpool-64/rpc\_buffers  
kmallo-1024 724 0 1024 182/0/1 biovec-64/sgpool-32  
kmallo-512 131 0 512 17/3/1 skbuff\_fclone\_cache/sgpool-16  
kmallo-256 738 0 256 50/21/1 filp/mnt\_cache/skbuff\_head\_cache/biovec-16/sgpool-8/arp\_cache/kiocb  
kmallo-128 429 0 128 17/4/1 bio/request\_sock\_TCP/flow\_cache/nfsd4\_stateids/nfs\_page  
kmallo-64 2478 0 64 43/20/1  
pid/fs\_cache/Acpi-ParseExt/Acpi-Operand/biovec-1/biovec-4/secpath\_cache/inet\_peer\_cache/tcp\_bind\_bucket/uid\_...  
kmallo-16 2821 0 16 12/0/1  
kmallo-8 416 0 8 1/0/1  
kmallo-32 659 0 32 6/2/1 Acpi-Namesp

Signed-off-by: Christoph Lameter <clameter@xxxxxxx>

Index: linux-2.6.21-rc2/fs/proc/proc\_misc.c

---

## [PATCH] SLUB The unqueued slab allocator V3

```
--- linux-2.6.21-rc2.orig/fs/proc/proc_misc.c 2007-02-27 20:59:12.000000000 -0800
+++ linux-2.6.21-rc2/fs/proc/proc_misc.c 2007-02-28 08:33:55.000000000 -0800
@@ -397,7 +397,7 @@
};
#endif

#ifndef CONFIG_SLAB
#ifdef CONFIG_SLAB || defined(CONFIG_SLUB)
extern struct seq_operations slabinfo_op;
extern ssize_t slabinfo_write(struct file *, const char __user *, size_t, loff_t *);
static int slabinfo_open(struct inode *inode, struct file *file)
@@ -407,7 +407,9 @@
static const struct file_operations proc_slabinfo_operations = {
.open = slabinfo_open,
.read = seq_read,
#ifdef CONFIG_SLAB
.write = slabinfo_write,
#endif
.llseek = seq_lseek,
.release = seq_release,
};
@@ -708,7 +710,7 @@
#endif
create_seq_entry("stat", 0, &proc_stat_operations);
create_seq_entry("interrupts", 0, &proc_interrupts_operations);
#ifndef CONFIG_SLAB
#ifdef CONFIG_SLAB || defined(CONFIG_SLUB)
create_seq_entry("slabinfo", S_IWUSR|S_IRUGO, &proc_slabinfo_operations);
#endif
#ifdef CONFIG_DEBUG_SLAB_LEAK
create_seq_entry("slab_allocators", 0, &proc_slabstats_operations);
Index: linux-2.6.21-rc2/include/linux/mm_types.h
=====
--- linux-2.6.21-rc2.orig/include/linux/mm_types.h 2007-02-27 20:59:12.000000000 -0800
+++ linux-2.6.21-rc2/include/linux/mm_types.h 2007-02-28 08:33:55.000000000 -0800
@@ -19,10 +19,16 @@
unsigned long flags; /* Atomic flags, some possibly
* updated asynchronously */
atomic_t _count; /* Usage count, see below. */
- atomic_t _mapcount; /* Count of ptes mapped in mms,
+ union {
+ atomic_t _mapcount; /* Count of ptes mapped in mms,
* to show when page is mapped
* & limit reverse map searches.
*/
+ struct { /* SLUB uses */
+ short unsigned int inuse;
+ short unsigned int offset;
+ };
+ };
union {
struct {
```

## [PATCH] SLUB The unqueued slab allocator V3

unsigned long private; /\* Mapping-private opaque data:

@@ -43,8 +49,15 @@

#if NR\_CPUS >= CONFIG\_SPLIT\_PTLOCK\_CPUS

spinlock\_t ptl;

#endif

+ struct { /\* SLUB uses \*/

+ struct page \*first\_page; /\* Compound pages \*/

+ struct kmem\_cache \*slab; /\* Pointer to slab \*/

+ };

+ };

+ union {

+ pgoff\_t index; /\* Our offset within mapping. \*/

+ void \*freelist; /\* SLUB: pointer to free object \*/

};

- pgoff\_t index; /\* Our offset within mapping. \*/

struct list\_head lru; /\* Pageout list, eg. active\_list

\* protected by zone->lru\_lock !

\*/

Index: linux-2.6.21-rc2/include/linux/slab.h

----- linux-2.6.21-rc2.orig/include/linux/slab.h 2007-02-27 20:59:12.000000000 -0800

+++ linux-2.6.21-rc2/include/linux/slab.h 2007-02-28 10:58:08.000000000 -0800

@@ -32,6 +32,7 @@

#define SLAB\_PANIC 0x00040000UL /\* Panic if kmem\_cache\_create() fails \*/

#define SLAB\_DESTROY\_BY\_RCU 0x00080000UL /\* Defer freeing slabs to RCU \*/

#define SLAB\_MEM\_SPREAD 0x00100000UL /\* Spread some memory over cpuset \*/

+#define SLAB\_TRACE 0x00200000UL /\* Trace allocations and frees \*/

/\* Flags passed to a constructor functions \*/

#define SLAB\_CTOR\_CONSTRUCTOR 0x001UL /\* If not set, then deconstructor \*/

@@ -94,9 +95,14 @@

\* the appropriate general cache at compile time.

\*/

-#ifndef CONFIG\_SLAB

+#if defined(CONFIG\_SLAB) || defined(CONFIG\_SLUB)

+#ifndef CONFIG\_SLUB

+#include <linux/slub\_def.h>

+#else

#include <linux/slab\_def.h>

+#endif /\* !CONFIG\_SLUB \*/

#else

+

/\*

\* Fallback definitions for an allocator not wanting to provide

\* its own optimized kmalloc definitions (like SLOB).

Index: linux-2.6.21-rc2/include/linux/slub\_def.h

--- /dev/null 1970-01-01 00:00:00.000000000 +0000

+++ linux-2.6.21-rc2/include/linux/slub\_def.h 2007-02-28 08:33:55.000000000 -0800

@@ -0,0 +1,168 @@

[PATCH] SLUB The unqueued slab allocator V3

## [PATCH] SLUB The unqueued slab allocator V3

```
+ #ifndef _LINUX_SLUB_DEF_H
+ #define _LINUX_SLUB_DEF_H
+
+ /*
+ * SLUB : A Slab allocator without object queues.
+ *
+ * (C) 2007 SGI, Christoph Lameter <clameter@xxxxxxx>
+ */
+ #include <linux/types.h>
+ #include <linux/gfp.h>
+ #include <linux/workqueue.h>
+
+ struct kmem_cache_node {
+ spinlock_t list_lock; /* Protect partial list and nr_partial */
+ unsigned long nr_partial;
+ atomic_long_t nr_slabs;
+ struct list_head partial;
+ };
+
+ /*
+ * Slab cache management.
+ */
+ struct kmem_cache {
+ int offset; /* Free pointer offset. */
+ unsigned int order;
+ unsigned long flags;
+ int size; /* Total size of an object */
+ int objects; /* Number of objects in slab */
+ struct kmem_cache_node local_node;
+ atomic_t refcount; /* Refcount for destroy */
+ void (*ctor)(void *, struct kmem_cache *, unsigned long);
+ void (*dtor)(void *, struct kmem_cache *, unsigned long);
+
+ int objsize; /* The size of an object that is in a chunk */
+ int inuse; /* Used portion of the chunk */
+ const char *name; /* Name (only for display!) */
+ char *aliases; /* Slabs merged into this one */
+ struct list_head list; /* List of slabs */
+ #ifdef CONFIG_SMP
+ struct mutex flushing;
+ atomic_t cpu_slabs; /*
+ * if >0 then flusher is scheduled. Also used
+ * to count remaining cpus if flushing
+ */
+ struct delayed_work flush;
+ #endif
+ #ifdef CONFIG_NUMA
+ struct kmem_cache_node *node[MAX_NUMNODES];
+ #endif
+ struct page *cpu_slab[NR_CPUS];
+ };
```

## [PATCH] SLUB The unqueued slab allocator V3

```
+
+/*
+ * Kmalloc subsystem.
+ */
+#define KMALLOC_SHIFT_LOW 3
+
+#define KMALLOC_SHIFT_HIGH 18
+
+#if L1_CACHE_BYTES <= 64
+#define KMALLOC_EXTRAS 2
+#define KMALLOC_EXTRA
+#else
+#define KMALLOC_EXTRAS 0
+#endif
+
+#define KMALLOC_NR_CACHES (KMALLOC_SHIFT_HIGH - KMALLOC_SHIFT_LOW \
+ + 1 + KMALLOC_EXTRAS)
+/*
+ * We keep the general caches in an array of slab caches that are used for
+ * 2^x bytes of allocations.
+ */
+extern struct kmem_cache kmalloc_caches[KMALLOC_NR_CACHES];
+
+/*
+ * Sorry that the following has to be that ugly but some versions of GCC
+ * have trouble with constant propagation and loops.
+ */
+static inline int kmalloc_index(int size)
+{
+ if (size <= 8) return 3;
+ if (size <= 16) return 4;
+ if (size <= 32) return 5;
+ if (size <= 64) return 6;
+#ifdef KMALLOC_EXTRA
+ if (size <= 96) return KMALLOC_SHIFT_HIGH + 1;
+#endif
+ if (size <= 128) return 7;
+#ifdef KMALLOC_EXTRA
+ if (size <= 192) return KMALLOC_SHIFT_HIGH + 2;
+#endif
+ if (size <= 256) return 8;
+ if (size <= 512) return 9;
+ if (size <= 1024) return 10;
+ if (size <= 2048) return 11;
+ if (size <= 4096) return 12;
+ if (size <= 8 * 1024) return 13;
+ if (size <= 16 * 1024) return 14;
+ if (size <= 32 * 1024) return 15;
+ if (size <= 64 * 1024) return 16;
+ if (size <= 128 * 1024) return 17;
+ if (size <= 256 * 1024) return 18;
```

## [PATCH] SLUB The unqueued slab allocator V3

```
+ return -1;
+}
+
+/*
+ * Find the slab cache for a given combination of allocation flags and size.
+ *
+ * This ought to end up with a global pointer to the right cache
+ * in kmalloc_caches.
+ */
+static inline struct kmem_cache *kmalloc_slab(size_t size)
+{
+ int index = kmalloc_index(size) - KMALLOC_SHIFT_LOW;
+
+ if (index < 0) {
+ /*
+ * Generate a link failure. Would be great if we could
+ * do something to stop the compile here.
+ */
+ extern void __kmalloc_size_too_large(void);
+ __kmalloc_size_too_large();
+ }
+ return &kmalloc_caches[index];
+}
+
+#ifdef CONFIG_ZONE_DMA
+#define SLUB_DMA __GFP_DMA
+#else
+/* Disable SLAB functionality */
+#define SLUB_DMA 0
+#endif
+
+static inline void *kmalloc(size_t size, gfp_t flags)
+{
+ if (__builtin_constant_p(size) && !(flags & SLUB_DMA)) {
+ struct kmem_cache *s = kmalloc_slab(size);
+
+ return kmem_cache_alloc(s, flags);
+ } else
+ return __kmalloc(size, flags);
+}
+
+static inline void *kzalloc(size_t size, gfp_t flags)
+{
+ if (__builtin_constant_p(size) && !(flags & SLUB_DMA)) {
+ struct kmem_cache *s = kmalloc_slab(size);
+
+ return kmem_cache_zalloc(s, flags);
+ } else
+ return __kzalloc(size, flags);
+}
+
```

## [PATCH] SLUB The unqueued slab allocator V3

```
+ #ifdef CONFIG_NUMA
+ extern void *__kmalloc_node(size_t size, gfp_t flags, int node);
+
+ static inline void *kmalloc_node(size_t size, gfp_t flags, int node)
+ {
+   if (__builtin_constant_p(size) && !(flags & SLUB_DMA)) {
+     struct kmem_cache *s = kmalloc_slab(size);
+
+     return kmem_cache_alloc_node(s, flags, node);
+   } else
+     return __kmalloc_node(size, flags, node);
+ }
+ #endif
+
+ #endif /* _LINUX_SLUB_DEF_H */
```

Index: linux-2.6.21-rc2/init/Kconfig

```
=====
--- linux-2.6.21-rc2.orig/init/Kconfig 2007-02-27 20:59:12.000000000 -0800
+++ linux-2.6.21-rc2/init/Kconfig 2007-02-28 09:38:24.000000000 -0800
@@ -458,15 +458,6 @@
option replaces shmem and tmpfs with the much simpler ramfs code,
which may be appropriate on small systems without swap.
```

-config SLAB

- default y
- bool "Use full SLAB allocator" if (EMBEDDED && !SMP && !SPARSEMEM)
- help
- Disabling this replaces the advanced SLAB allocator and
- kmalloc support with the drastically simpler SLOB allocator.
- SLOB is more space efficient but does not scale well and is
- more susceptible to fragmentation.

-  
config VM\_EVENT\_COUNTERS

default y

bool "Enable VM event counters for /proc/vmstat" if EMBEDDED

@@ -476,6 +467,45 @@

on EMBEDDED systems. /proc/vmstat will only show page counts  
if VM event counters are disabled.

+choice

- + prompt "Choose SLAB allocator"
- + default SLAB
- + help
- + This option allows to select a slab allocator.

+

+config SLAB

+ bool "SLAB"

+ help

- + The regular slab allocator that is established and known to work
- + well in all environments. It organizes cache hot objects in
- + per cpu and per node queues. SLAB is the default choice for

## [PATCH] SLUB The unqueued slab allocator V3

```
+ slab allocator.
+
+config SLUB
+ depends on EXPERIMENTAL
+ bool "SLUB (Unqueued Allocator)"
+ help
+ SLUB is a slab allocator that minimizes cache line usage
+ instead of managing queues of cached objects (SLAB approach).
+ Per cpu caching is realized using slabs of objects instead
+ of queues of objects. SLUB can use memory in the most efficient
+ way and has enhanced diagnostics.
+
+config SLOB
+#
+# SLOB does not support SMP because SLAB_DESTROY_BY_RCU is not support.
+#
+ depends on EMBEDDED && !SMP
+ bool "SLOB (Simple Allocator)"
+ help
+ SLOB replaces the SLAB allocator with a drastically simpler
+ allocator. SLOB is more space efficient than SLAB but does not
+ scale well (single lock for all operations) and is more susceptible
+ to fragmentation. SLOB is a great choice to reduce
+ memory usage and code size.
+
+endchoice
+
endmenu # General setup

config RT_MUTEXES
@@ -491,10 +521,6 @@
default 0 if BASE_FULL
default 1 if !BASE_FULL

- config SLOB
- default !SLAB
- bool
-
menu "Loadable module support"

config MODULES
Index: linux-2.6.21-rc2/mm/Makefile
=====
--- linux-2.6.21-rc2.orig/mm/Makefile 2007-02-27 20:59:12.000000000 -0800
+++ linux-2.6.21-rc2/mm/Makefile 2007-02-28 08:33:55.000000000 -0800
@@ -25,6 +25,7 @@
obj-$(CONFIG_TINY_SHMEM) += tiny-shmem.o
obj-$(CONFIG_SLOB) += slob.o
obj-$(CONFIG_SLAB) += slab.o
+obj-$(CONFIG_SLUB) += slub.o
obj-$(CONFIG_MEMORY_HOTPLUG) += memory_hotplug.o
```

## [PATCH] SLUB The unqueued slab allocator V3

obj-\$(CONFIG\_FS\_XIP) += filemap\_xip.o  
obj-\$(CONFIG\_MIGRATION) += migrate.o  
Index: linux-2.6.21-rc2/mm/slub.c

```
=====
--- /dev/null 1970-01-01 00:00:00.000000000 +0000
+++ linux-2.6.21-rc2/mm/slub.c 2007-02-28 10:58:08.000000000 -0800
@@ -0,0 +1,2169 @@
+/*
+ * SLUB: A slab allocator that limits cache line use instead of queuing
+ * objects in per cpu and per node lists.
+ *
+ * The allocator synchronizes using per slab locks and only
+ * uses a centralized lock to manage a pool of partial slabs.
+ *
+ * (C) 2007 SGI, Christoph Lameter <clameter@xxxxxxx>
+ */
+
+#include <linux/mm.h>
+#include <linux/module.h>
+#include <linux/bit_spinlock.h>
+#include <linux/interrupt.h>
+#include <linux/bitops.h>
+#include <linux/slab.h>
+#include <linux/seq_file.h>
+#include <linux/cpu.h>
+#include <linux/cpuset.h>
+#include <linux/mempolicy.h>
+#include <linux/ctype.h>
+
+/*
+ * Lock order:
+ * 1. slab_lock(page)
+ * 2. slab->list_lock
+ *
+ * SLUB assigns one slab for allocation to each processor.
+ * Allocations only occur from these slabs called cpu slabs.
+ *
+ * If a cpu slab exists then a workqueue thread checks every 10
+ * seconds if the cpu slab is still in use. The cpu slab is pushed back
+ * to the list if inactive [only needed for SMP].
+ *
+ * Slabs with free elements are kept on a partial list.
+ * There is no list for full slabs. If an object in a full slab is
+ * freed then the slab will show up again on the partial lists.
+ * Otherwise there is no need to track full slabs (but we keep a counter).
+ *
+ * Slabs are freed when they become empty. Teardown and setup is
+ * minimal so we rely on the page allocators per cpu caches for
+ * fast frees and allocs.
+ *
+ * Overloading of page flags that are otherwise used for LRU management.

```

## [PATCH] SLUB The unqueued slab allocator V3

```
+ *
+ * PageActive The slab is used as a cpu cache. Allocations
+ * may be performed from the slab. The slab is not
+ * on a partial list.
+ *
+ * PageReferenced The per cpu slab was used recently. This is used
+ * to push back per cpu slabs if they are unused
+ * for a longer time period.
+ *
+ * PageError Slab requires special handling due to debug
+ * options set or a single page slab. This moves
+ * slab handling out of the fast path.
+ */
+
+ /*
+ * Flags from the regular SLAB that SLUB does not support:
+ */
+#define SLUB_UNIMPLEMENTED (SLAB_DEBUG_INITIAL)
+
+#define DEBUG_DEFAULT_FLAGS (SLAB_DEBUG_FREE | SLAB_RED_ZONE | \
+ SLAB_POISON)
+ /*
+ * Set of flags that will prevent slab merging
+ */
+#define SLUB_NEVER_MERGE (SLAB_RED_ZONE | SLAB_POISON | SLAB_STORE_USER | \
+ SLAB_TRACE)
+
+#define SLUB_MERGE_SAME (SLAB_DEBUG_FREE | SLAB_DESTROY_BY_RCU | \
+ SLAB_RECLAIM_ACCOUNT | SLAB_CACHE_DMA)
+
+#ifndef ARCH_KMALLOC_MINALIGN
+#define ARCH_KMALLOC_MINALIGN sizeof(void *)
+#endif
+
+#ifndef ARCH_SLAB_MINALIGN
+#define ARCH_SLAB_MINALIGN sizeof(void *)
+#endif
+
+static int kmem_size = sizeof(struct kmem_cache);
+
+ /*
+ * Forward declarations
+ */
+static void register_slab(struct kmem_cache *s);
+static void unregister_slab(struct kmem_cache *s);
+
+#ifdef CONFIG_SMP
+static struct notifier_block slab_notifier;
+#endif
+
+static enum {
```

## [PATCH] SLUB The unqueued slab allocator V3

```
+ DOWN, /* No slab functionality available */
+ PARTIAL, /* kmem_cache_open() works but kmalloc does not */
+ UP /* Everything works */
+} slab_state = DOWN;
+
+int slab_is_available(void) {
+ return slab_state == UP;
+}
+
+/* A list of all slab caches on the system */
+static DECLARE_RWSEM(slabstat_sem);
+LIST_HEAD(slab_caches);
+
+/**
+ * Core slab cache functions
+ */
+
+struct kmem_cache_node *get_node(struct kmem_cache *s, int node)
+{
+#ifdef CONFIG_NUMA
+ return s->node[node];
+#else
+ return &s->local_node;
+#endif
+}
+
+/*
+ * Object debugging
+ */
+static void print_section(char *text, u8 *addr, unsigned int length)
+{
+ int i, offset;
+ int newline = 1;
+ char ascii[17];
+
+ if (length > 128)
+ length = 128;
+ ascii[16] = 0;
+
+ for (i = 0; i < length; i++) {
+ if (newline) {
+ printk(KERN_ERR "%10s %p: ", text, addr + i);
+ newline = 0;
+ }
+ printk(" %02x", addr[i]);
+ offset = i % 16;
+ ascii[offset] = isgraph(addr[i]) ? addr[i] : '.';
+ if (offset == 15) {
+ printk(" %s\n", ascii);
+ newline = 1;
+ }
+ }
+}
```

## [PATCH] SLUB The unqueued slab allocator V3

```
+ }
+ if (!newline) {
+ i %= 16;
+ while (i < 16) {
+ printk(" ");
+ ascii[i] = ' ';
+ i++;
+ }
+ printk(" %s\n", ascii);
+ }
+}
+
+/*
+ * Slow version of get and set free pointer.
+ *
+ * This requires touching the cache lines of kmem_cache.
+ * The offset can also be obtained from the page. In that
+ * case it is in the cacheline that we already need to touch.
+ */
+static void *get_freepointer(struct kmem_cache *s, void *object)
+{
+ void **p = object;
+
+ return p[s->offset];
+}
+
+static void set_freepointer(struct kmem_cache *s, void *object, void *fp)
+{
+ void **p = object;
+
+ p[s->offset] = fp;
+}
+
+/*
+ * Tracking user of a slab.
+ */
+static void *get_track(struct kmem_cache *s, void *object, int alloc)
+{
+ void **p = object + s->inuse + sizeof(void *);
+
+ return p[alloc];
+}
+
+static void set_track(struct kmem_cache *s, void *object,
+ int alloc, void *addr)
+{
+ void **p = object + s->inuse + sizeof(void *);
+
+ p[alloc] = addr;
+}
+
```

## [PATCH] SLUB The unqueued slab allocator V3

```
+ #define set_tracking(__s, __o, __a) set_track(__s, __o, __a, \
+ __builtin_return_address(0))
+
+ static void init_tracking(struct kmem_cache *s, void *object)
+ {
+   if (s->flags & SLAB_STORE_USER) {
+     set_track(s, object, 0, NULL);
+     set_track(s, object, 1, NULL);
+   }
+ }
+
+ static void print_trailer(struct kmem_cache *s, u8 *p)
+ {
+   unsigned int off;
+
+   if (s->offset)
+     off = s->offset + sizeof(void *);
+   else
+     off = s->inuse;
+
+   if (s->flags & SLAB_RED_ZONE)
+     print_section("Redzone", p + s->objsize,
+ s->inuse - s->objsize);
+
+   printk(KERN_ERR "FreePointer %p: %p\n", p + s->offset,
+ get_freepointer(s, p));
+
+   if (s->flags & SLAB_STORE_USER) {
+     printk(KERN_ERR "Last Allocate from %p. Last Free from %p\n",
+ get_track(s, p, 0), get_track(s, p, 1));
+     off += 2 * sizeof(void *);
+   }
+
+   if (off != s->size)
+     /* Beginning of the filler is the free pointer */
+     print_section("Filler", p + off, s->size - off);
+ }
+
+ static void object_err(struct kmem_cache *s, struct page *page,
+ u8 *object, char *reason)
+ {
+   u8 *addr = page_address(page);
+
+   printk(KERN_ERR "*** SLUB: %s in %s@%p Slab %p\n",
+ reason, s->name, object, page);
+   printk(KERN_ERR " offset=%u flags=%04lx inuse=%u freelist=%p\n",
+ (int)(object - addr), page->flags, page->inuse, page->freelist);
+   if (object > addr + 16)
+     print_section("Bytes b4", object - 16, 16);
+   print_section("Object", object, s->objsize);
+   print_trailer(s, object);
+ }
```

## [PATCH] SLUB The unqueued slab allocator V3

```
+ dump_stack();
+}
+
+static void init_object(struct kmem_cache *s, void *object, int active)
+{
+ u8 *p = object;
+
+ if (s->objects == 1)
+ return;
+
+ if (s->flags & SLAB_POISON) {
+ memset(p, POISON_FREE, s->objsize - 1);
+ p[s->objsize - 1] = POISON_END;
+ }
+
+ if (s->flags & SLAB_RED_ZONE)
+ memset(p + s->objsize,
+ active ? RED_ACTIVE : RED_INACTIVE,
+ s->inuse - s->objsize);
+}
+
+static int check_bytes(u8 *start, unsigned int value, unsigned int bytes)
+{
+ while (bytes) {
+ if (*start != (u8)value)
+ return 0;
+ start++;
+ bytes--;
+ }
+ return 1;
+}
+
+static int check_valid_pointer(struct kmem_cache *s, struct page *page,
+ void *object)
+{
+ void *base;
+
+ if (!object)
+ return 1;
+
+ base = page_address(page);
+ if (object < base || object >= base + s->objects * s->size ||
+ (object - base) % s->size) {
+ return 0;
+ }
+
+ return 1;
+}
+
+/*
```

## [PATCH] SLUB The unqueued slab allocator V3

```
+ * Object layout:
+ *
+ * object address
+ * Bytes of the object to be managed.
+ * If the freepointer may overlay the object then the free
+ * pointer is the first word of the object.
+ * object + s->objsize
+ * Padding to reach word boundary. This is also used for Redzoning.
+ * Padding is extended to word size if Redzoning is enabled
+ * and objsize == inuse.
+ * object + s->inuse
+ * A. Free pointer (if we cannot overwrite object on free)
+ * B. Tracking data for SLAB_STORE_USER
+ * C. Padding to reach required alignment boundary
+ * object + s->size
+ *
+ * If slabcaches are merged then the objsize and inuse boundaries are to be ignored.
+ */
+static int check_object(struct kmem_cache *s, struct page *page,
+ void *object, int active)
+{
+ u8 *p = object;
+ u8 *endobject = object + s->objsize;
+
+ /* Offset of first byte after free pointer */
+ unsigned long off = s->inuse;
+
+ if (s->offset)
+ off += sizeof(void *);
+
+ /* Single object slabs do not get policed */
+ if (s->objects == 1)
+ return 1;
+
+ if (s->flags & SLAB_RED_ZONE) {
+ if (!check_bytes(endobject,
+ active ? RED_ACTIVE : RED_INACTIVE,
+ s->inuse - s->objsize)) {
+ object_err(s, page, object,
+ active ? "Redzone Active check fails" :
+ "Redzone Inactive check fails");
+ return 0;
+ }
+ } else
+ if ((s->flags & SLAB_POISON) &&
+ s->objsize < s->inuse &&
+ !check_bytes(endobject, POISON_INUSE, s->inuse - s->objsize))
+ object_err(s, page, p, "Alignment Filler check fails");
+
+ if (s->flags & SLAB_POISON) {
+ if (!active && (!check_bytes(p, POISON_FREE, s->objsize - 1) ||
```

## [PATCH] SLUB The unqueued slab allocator V3

```
+ p[s->objsize - 1] != POISON_END)) {
+ object_err(s, page, p, "Poison");
+ return 0;
+ }
+ if (s->size > off && !check_bytes(p + off,
+ POISON_INUSE, s->size - off))
+ object_err(s, page, p,
+ "Interobject Filler check fails");
+ }
+
+ if (s->offset == 0 && active)
+ /*
+ * Object and freepointer overlap. Cannot check
+ * if object is allocated.
+ */
+ return 1;
+
+ /* Check free pointer validity */
+ if (!check_valid_pointer(s, page, get_freepointer(s, p))) {
+ object_err(s, page, p, "Freepointer corrupt");
+ /*
+ * No choice but to zap it. This may cause
+ * another error because the object count
+ * is now wrong.
+ */
+ set_freepointer(s, p, NULL);
+ return 0;
+ }
+ return 1;
+}
+
+static int check_slab(struct kmem_cache *s, struct page *page)
+{
+ if (!PageSlab(page)) {
+ printk(KERN_CRIT "SLUB: %s Not a valid slab page @%p flags=%lx"
+ " mapping=%p count=%d\n",
+ s->name, page, page->flags, page->mapping,
+ page_count(page));
+ return 0;
+ }
+ if (page->offset != s->offset) {
+ printk(KERN_CRIT "SLUB: %s Corrupted offset %u in slab @%p"
+ " flags=%lx mapping=%p count=%d\n",
+ s->name, page->offset, page, page->flags,
+ page->mapping, page_count(page));
+ return 0;
+ }
+ if (page->inuse > s->objects) {
+ printk(KERN_CRIT "SLUB: %s Inuse %u > max %u in slab page @%p"
+ " flags=%lx mapping=%p count=%d\n",
+ s->name, page->inuse, s->objects, page, page->flags,
```

## [PATCH] SLUB The unqueued slab allocator V3

```
+ page->mapping, page_count(page));
+ return 0;
+ }
+ return 1;
+}
+
+/*
+ * Determine if a certain object on a page is on the freelist and
+ * therefore free. Must hold the slab lock for cpu slabs to
+ * guarantee that the chains are consistent.
+ */
+static int on_freelist(struct kmem_cache *s, struct page *page, void *search)
+{
+ int nr = 0;
+ void *fp = page->freelist;
+ void *object = NULL;
+
+ if (s->objects == 1)
+ return 0;
+
+ while (fp && nr <= s->objects) {
+ if (fp == search)
+ return 1;
+ if (!check_valid_pointer(s, page, fp)) {
+ if (object) {
+ object_err(s, page, object, "Freechain corrupt");
+ set_freepointer(s, object, NULL);
+ break;
+ } else {
+ printk(KERN_ERR "SLUB: %s slab %p freepointer %p corrupted.\n",
+ s->name, page, fp);
+ dump_stack();
+ page->freelist = NULL;
+ page->inuse = s->objects;
+ return 0;
+ }
+ break;
+ }
+ object = fp;
+ fp = get_freepointer(s, object);
+ nr++;
+ }
+
+ if (page->inuse != s->objects - nr) {
+ printk(KERN_CRIT "slab %s: page %p wrong object count."
+ " counter is %d but counted were %d\n",
+ s->name, page, page->inuse,
+ s->objects - nr);
+ page->inuse = s->objects - nr;
+ }
+ return 0;
+}
```

## [PATCH] SLUB The unqueued slab allocator V3

```
+}
+
+static int alloc_object_checks(struct kmem_cache *s, struct page *page,
+ void *object)
+{
+ if (!check_slab(s, page))
+ goto bad;
+
+
+ if (object && !on_freelist(s, page, object)) {
+ printk(KERN_ERR "SLAB: %s Object %p@%p already allocated.\n",
+ s->name, object, page);
+ goto dump;
+ }
+
+
+ if (!check_valid_pointer(s, page, object)) {
+ object_err(s, page, object, "Freelist Pointer check fails");
+ goto dump;
+ }
+
+
+ if (!object)
+ return 1;
+
+
+ if (!check_object(s, page, object, 0))
+ goto bad;
+ init_object(s, object, 1);
+
+
+ if (s->flags & SLAB_TRACE) {
+ printk("SLUB-Trace %s alloc object=%p slab=%p inuse=%d"
+ " freelist=%p\n",
+ s->name, object, page, page->inuse,
+ page->freelist);
+ dump_stack();
+ }
+ return 1;
+dump:
+ dump_stack();
+bad:
+ /* Mark slab full */
+ page->inuse = s->objects;
+ page->freelist = NULL;
+ return 0;
+}
+
+
+static int free_object_checks(struct kmem_cache *s, struct page *page, void *object)
+{
+ if (!check_slab(s, page)) {
+ goto fail;
+ }
+
+
+ if (!check_valid_pointer(s, page, object)) {
+ printk(KERN_ERR "SLUB: %s slab %p invalid free pointer %p\n",
```

## [PATCH] SLUB The unqueued slab allocator V3

```
+ s->name, page, object);
+ goto fail;
+ }
+
+ if (on_freelist(s, page, object)) {
+ printk(KERN_CRIT "SLUB: %s slab %p object %p already free.\n",
+ s->name, page, object);
+ goto fail;
+ }
+
+ if (!check_object(s, page, object, 1))
+ return 0;
+
+ if (unlikely(s != page->slab)) {
+ if (!PageSlab(page))
+ printk(KERN_CRIT "slab_free %s size %d: attempt to"
+ "free object(%p) outside of slab.\n",
+ s->name, s->size, object);
+ else
+ if (!page->slab)
+ printk(KERN_CRIT
+ "slab_free : no slab(NULL) for object %p.\n",
+ object);
+ else
+ printk(KERN_CRIT "slab_free %s(%d): object at %p"
+ " belongs to slab %s(%d)\n",
+ s->name, s->size, object,
+ page->slab->name, page->slab->size);
+ goto fail;
+ }
+ if (s->flags & SLAB_TRACE) {
+ printk("SLUB-Trace %s free object=%p slab=%p"
+ "inuse=%d freelist=%p\n",
+ s->name, object, page, page->inuse,
+ page->freelist);
+ print_section("SLUB-Trace", object, s->objsize);
+ dump_stack();
+ }
+ init_object(s, object, 0);
+ return 1;
+fail:
+ dump_stack();
+ return 0;
+}
+
+/*
+ * Slab allocation and freeing
+ */
+static struct page *allocate_slab(struct kmem_cache *s, gfp_t flags, int node)
+{
+ struct page * page;
```

## [PATCH] SLUB The unqueued slab allocator V3

```
+ int pages = 1 << s->order;
+
+ if (s->order)
+ flags |= __GFP_COMP;
+
+ if (s->flags & SLUB_DMA)
+ flags |= GFP_DMA;
+
+ if (node == -1)
+ page = alloc_pages(flags, s->order);
+ else
+ page = alloc_pages_node(node, flags, s->order);
+
+ if (!page)
+ return NULL;
+
+ mod_zone_page_state(page_zone(page),
+ (s->flags & SLAB_RECLAIM_ACCOUNT) ?
+ NR_SLAB_RECLAIMABLE : NR_SLAB_UNRECLAIMABLE,
+ pages);
+
+ if (unlikely(s->ctor)) {
+ void *start = page_address(page);
+ void *end = start + (pages << PAGE_SHIFT);
+ void *p;
+ int mode = 1;
+
+ if (!(flags & __GFP_WAIT))
+ mode |= SLAB_CTOR_ATOMIC;
+
+ for (p = start; p <= end - s->size; p += s->size)
+ s->ctor(p, s, mode);
+ }
+ return page;
+}
+
+static struct page *new_slab(struct kmem_cache *s, gfp_t flags, int node)
+{
+ struct page *page;
+ struct kmem_cache_node *n;
+
+ BUG_ON(flags & ~(GFP_DMA | GFP_LEVEL_MASK | __GFP_NO_GROW));
+ if (flags & __GFP_NO_GROW)
+ return NULL;
+
+ if (flags & __GFP_WAIT)
+ local_irq_enable();
+
+ page = allocate_slab(s, flags & GFP_LEVEL_MASK, node);
+ if (!page)
+ goto out;
```

## [PATCH] SLUB The unqueued slab allocator V3

```
+
+ n = get_node(s, page_to_nid(page));
+ if (n)
+ atomic_long_inc(&n->nr_slabs);
+ page->offset = s->offset;
+ page->slab = s;
+ page->flags |= 1 << PG_slab;
+ if (s->flags & (SLAB_DEBUG_FREE | SLAB_RED_ZONE | SLAB_POISON |
+ SLAB_STORE_USER | SLAB_TRACE) ||
+ s->objects == 1)
+ page->flags |= 1 << PG_error;
+
+ if (s->objects > 1) {
+ void *start = page_address(page);
+ void *end = start + s->objects * s->size;
+ void **last = start;
+ void *p = start + s->size;
+
+ if (unlikely(s->flags & SLAB_POISON))
+ memset(start, POISON_INUSE, PAGE_SIZE << s->order);
+ while (p < end) {
+ if (PageError(page)) {
+ init_object(s, last, 0);
+ init_tracking(s, last);
+ }
+ last[s->offset] = p;
+ last = p;
+ p += s->size;
+ }
+ last[s->offset] = NULL;
+ page->freelist = start;
+ page->inuse = 0;
+ if (PageError(page)) {
+ init_object(s, last, 0);
+ init_tracking(s, last);
+ }
+ }
+
+out:
+ if (flags & __GFP_WAIT)
+ local_irq_disable();
+ return page;
+}
+
+static void __free_slab(struct kmem_cache *s, struct page *page)
+{
+ int pages = 1 << s->order;
+
+ if (unlikely(PageError(page))) {
+ void *start = page_address(page);
```

## [PATCH] SLUB The unqueued slab allocator V3

```
+ void *end = start + (pages << PAGE_SHIFT);
+ void *p;
+
+ for (p = start; p <= end - s->size; p += s->size) {
+ if (s->dtor)
+ s->dtor(p, s, 0);
+ else
+ check_object(s, page, p, 0);
+ }
+ if ((s->flags & SLAB_POISON) &&
+ check_bytes(end, POISON_INUSE,
+ (PAGE_SIZE << s->order) - (end - start)))
+ object_err(s, page, p, "Slab End fill check fails");
+ }
+
+ mod_zone_page_state(page_zone(page),
+ (s->flags & SLAB_RECLAIM_ACCOUNT) ?
+ NR_SLAB_RECLAIMABLE : NR_SLAB_UNRECLAIMABLE,
+ - pages);
+
+ __free_pages(page, s->order);
+}
+
+static void rcu_free_slab(struct rcu_head *h)
+{
+ struct page *page;
+ struct kmem_cache *s;
+
+ page = container_of((struct list_head *)h, struct page, lru);
+ s = (struct kmem_cache *)page->mapping;
+ page->mapping = NULL;
+ __free_slab(s, page);
+}
+
+static void free_slab(struct kmem_cache *s, struct page *page)
+{
+ if (unlikely(s->flags & SLAB_DESTROY_BY_RCU)) {
+ /*
+ * RCU free overloads the RCU head over the LRU
+ */
+ struct rcu_head *head = (void *)&page->lru;
+
+ page->mapping = (void *)s;
+ call_rcu(head, rcu_free_slab);
+ } else
+ __free_slab(s, page);
+}
+
+static void discard_slab(struct kmem_cache *s, struct page *page)
+{
+ struct kmem_cache_node *n = get_node(s, page_to_nid(page));
```

## [PATCH] SLUB The unqueued slab allocator V3

```
+
+ atomic_long_dec(&n->nr_slabs);
+
+ page->mapping = NULL;
+ reset_page_mapcount(page);
+ page->flags &= ~(1 << PG_slab | 1 << PG_error);
+ free_slab(s, page);
+}
+
+/*
+ * Per slab locking using the pagelock
+ */
+static __always_inline void slab_lock(struct page *page)
+{
+#ifdef CONFIG_SMP
+ bit_spin_lock(PG_locked, &page->flags);
+#endif
+}
+
+static __always_inline void slab_unlock(struct page *page)
+{
+#ifdef CONFIG_SMP
+ bit_spin_unlock(PG_locked, &page->flags);
+#endif
+}
+
+static __always_inline int slab_trylock(struct page *page)
+{
+ int rc = 1;
+#ifdef CONFIG_SMP
+ rc = bit_spin_trylock(PG_locked, &page->flags);
+#endif
+ return rc;
+}
+
+/*
+ * Management of partially allocated slabs
+ */
+static void __always_inline add_partial(struct kmem_cache *s, struct page *page)
+{
+ struct kmem_cache_node *n = get_node(s, page_to_nid(page));
+
+ spin_lock(&n->list_lock);
+ n->nr_partial++;
+ list_add_tail(&page->lru, &n->partial);
+ spin_unlock(&n->list_lock);
+}
+
+static void __always_inline remove_partial(struct kmem_cache *s,
+ struct page *page)
+{
```

## [PATCH] SLUB The unqueued slab allocator V3

```
+ struct kmem_cache_node *n = get_node(s, page_to_nid(page));
+
+ spin_lock(&n->list_lock);
+ list_del(&page->lru);
+ n->nr_partial--;
+ spin_unlock(&n->list_lock);
+}
+
+/*
+ * Lock page and remove it from the partial list
+ *
+ * Must hold list_lock
+ */
+static __always_inline int lock_and_del_slab(struct kmem_cache_node *n,
+ struct page *page)
+{
+ if (slab_trylock(page)) {
+ list_del(&page->lru);
+ n->nr_partial--;
+ return 1;
+ }
+ return 0;
+}
+
+/*
+ * Try to get a partial slab from a specific node
+ */
+static struct page *get_partial_node(struct kmem_cache_node *n)
+{
+ struct page *page;
+
+ /*
+ * Racy check. If we mistakenly see no partial slabs then we
+ * just allocate an empty slab. If we mistakenly try to get a
+ * partial slab then get_partials() will return NULL.
+ */
+ if (!n || !n->nr_partial)
+ return NULL;
+
+ spin_lock(&n->list_lock);
+ list_for_each_entry(page, &n->partial, lru)
+ if (lock_and_del_slab(n, page))
+ goto out;
+ page = NULL;
+out:
+ spin_unlock(&n->list_lock);
+ return page;
+}
+
+/*
+ * Get a page from somewhere. Search in increasing NUMA
```

## [PATCH] SLUB The unqueued slab allocator V3

```
+ * distance.
+ */
+static struct page *get_any_partial(struct kmem_cache *s, gfp_t flags)
+{
+ #ifdef CONFIG_NUMA
+ struct zonelist *zonelist = &NODE_DATA(slab_node(current->mempolicy))
+ ->node_zonelist[gfp_zone(flags)];
+ struct zone **z;
+ struct page *page;
+
+ for (z = zonelist->zones; *z; z++) {
+ struct kmem_cache_node *n;
+
+ n = get_node(s, zone_to_nid(*z));
+
+ if (n && cpuset_zone_allowed_hardwall(*z, flags) &&
+ n->nr_partial) {
+ page = get_partial_node(n);
+ if (page)
+ return page;
+ }
+ }
+ #endif
+ return NULL;
+}
+
+ /*
+ * Get a partial page, lock it and return it.
+ */
+static struct page *get_partial(struct kmem_cache *s, gfp_t flags, int node)
+{
+ struct page *page;
+ int searchnode = (node == -1) ? numa_node_id() : node;
+
+ page = get_partial_node(get_node(s, searchnode));
+ if (page || (flags & __GFP_THISNODE))
+ return page;
+
+ return get_any_partial(s, flags);
+}
+
+ /*
+ * Move a page back to the lists.
+ *
+ * Must be called with the slab lock held.
+ *
+ * On exit the slab lock will have been dropped.
+ */
+static void __always_inline putback_slab(struct kmem_cache *s, struct page *page)
+{
+ if (page->inuse) {
```

## [PATCH] SLUB The unqueued slab allocator V3

```
+ if (page->inuse < s->objects)
+ add_partial(s, page);
+ slab_unlock(page);
+ } else {
+ slab_unlock(page);
+ discard_slab(s, page);
+ }
+}
+
+/*
+ * Remove the cpu slab
+ */
+static void __always_inline deactivate_slab(struct kmem_cache *s,
+ struct page *page, int cpu)
+{
+ s->cpu_slab[cpu] = NULL;
+ ClearPageActive(page);
+ ClearPageReferenced(page);
+
+ putback_slab(s, page);
+}
+
+static void flush_slab(struct kmem_cache *s, struct page *page, int cpu)
+{
+ slab_lock(page);
+ deactivate_slab(s, page, cpu);
+}
+
+/*
+ * Flush cpu slab.
+ * Called from IPI handler with interrupts disabled.
+ */
+static void __flush_cpu_slab(struct kmem_cache *s, int cpu)
+{
+ struct page *page = s->cpu_slab[cpu];
+
+ if (likely(page))
+ flush_slab(s, page, cpu);
+}
+
+static void flush_cpu_slab(void *d)
+{
+ struct kmem_cache *s = d;
+ int cpu = smp_processor_id();
+
+ __flush_cpu_slab(s, cpu);
+}
+
+#ifdef CONFIG_SMP
+/*
+ * Called from IPI to check and flush cpu slabs.
```

## [PATCH] SLUB The unqueued slab allocator V3

```
+ */
+static void check_flush_cpu_slab(void *private)
+{
+ struct kmem_cache *s = private;
+ int cpu = smp_processor_id();
+ struct page *page = s->cpu_slab[cpu];
+
+ if (page) {
+ if (!TestClearPageReferenced(page))
+ return;
+ flush_slab(s, page, cpu);
+ }
+ atomic_dec(&s->cpu_slabs);
+}
+
+/*
+ * Called from eventd
+ */
+static void flusher(struct work_struct *w)
+{
+ struct kmem_cache *s = container_of(w, struct kmem_cache, flush.work);
+
+ if (!mutex_trylock(&s->flushing))
+ return;
+
+ atomic_set(&s->cpu_slabs, num_online_cpus());
+ on_each_cpu(check_flush_cpu_slab, s, 1, 1);
+ if (atomic_read(&s->cpu_slabs))
+ schedule_delayed_work(&s->flush, 30 * HZ);
+ mutex_unlock(&s->flushing);
+}
+
+static void flush_all(struct kmem_cache *s)
+{
+ if (atomic_read(&s->cpu_slabs)) {
+ mutex_lock(&s->flushing);
+ cancel_delayed_work(&s->flush);
+ atomic_set(&s->cpu_slabs, 0);
+ on_each_cpu(flush_cpu_slab, s, 1, 1);
+ mutex_unlock(&s->flushing);
+ }
+}
+
+#else
+static void flush_all(struct kmem_cache *s)
+{
+ unsigned long flags;
+
+ local_irq_save(flags);
+ flush_cpu_slab(s);
+ local_irq_restore(flags);
+}
+}
```

## [PATCH] SLUB The unqueued slab allocator V3

```
+#endif
+
+static __always_inline void *slab_alloc(struct kmem_cache *s,
+ gfp_t gfpflags, int node)
+{
+ struct page *page;
+ void **object;
+ unsigned long flags;
+ int cpu;
+
+ local_irq_save(flags);
+ cpu = smp_processor_id();
+ page = s->cpu_slab[cpu];
+ if (!page)
+ goto new_slab;
+
+ slab_lock(page);
+ if (unlikely(node != -1 && page_to_nid(page) != node))
+ goto another_slab;
+ if (unlikely(!page->freelist))
+ goto another_slab;
+redo:
+ object = page->freelist;
+ if (unlikely(PageError(page))) {
+ if (!alloc_object_checks(s, page, object))
+ goto another_slab;
+ if (s->flags & SLAB_STORE_USER)
+ set_tracking(s, object, 0);
+ }
+ page->inuse++;
+ page->freelist = object[page->offset];
+ SetPageReferenced(page);
+ slab_unlock(page);
+ local_irq_restore(flags);
+ return object;
+
+another_slab:
+ deactivate_slab(s, page, cpu);
+
+new_slab:
+ page = get_partial(s, gfpflags, node);
+ if (page)
+ goto gotpage;
+
+ page = new_slab(s, gfpflags, node);
+ if (!page) {
+ local_irq_restore(flags);
+ return NULL;
+ }
+
+ if (unlikely(s->objects == 1)) {
```

## [PATCH] SLUB The unqueued slab allocator V3

```
+ local_irq_restore(flags);
+ return page_address(page);
+ }
+
+ slab_lock(page);
+
+gotpage:
+ if (unlikely(s->cpu_slab[cpu])) {
+ slab_unlock(page);
+ discard_slab(s, page);
+ page = s->cpu_slab[cpu];
+ slab_lock(page);
+ goto redo;
+ }
+
+ s->cpu_slab[cpu] = page;
+ SetPageActive(page);
+
+#ifdef CONFIG_SMP
+ if (!atomic_read(&s->cpu_slabs) && keventd_up()) {
+ atomic_inc(&s->cpu_slabs);
+ schedule_delayed_work(&s->flush, 30 * HZ);
+ }
+#endif
+ goto redo;
+}
+
+void *kmem_cache_alloc(struct kmem_cache *s, gfp_t gfpflags)
+{
+ return slab_alloc(s, gfpflags, -1);
+}
+EXPORT_SYMBOL(kmem_cache_alloc);
+
+#ifdef CONFIG_NUMA
+void *kmem_cache_alloc_node(struct kmem_cache *s, gfp_t gfpflags, int node)
+{
+ return slab_alloc(s, gfpflags, node);
+}
+EXPORT_SYMBOL(kmem_cache_alloc_node);
+#endif
+
+void kmem_cache_free(struct kmem_cache *s, void *x)
+{
+ struct page * page;
+ void *prior;
+ void **object = (void *)x;
+ unsigned long flags;
+
+ if (!object)
+ return;
+}
```

## [PATCH] SLUB The unqueued slab allocator V3

```
+ page = virt_to_page(x);
+
+ if (unlikely(PageCompound(page)))
+ page = page->first_page;
+
+ if (!s)
+ s = page->slab;
+
+ local_irq_save(flags);
+
+ if (unlikely(PageError(page)) && s->objects == 1)
+ goto single_object_slab;
+
+ slab_lock(page);
+
+ if (unlikely(PageError(page))) {
+ if (!free_object_checks(s, page, x))
+ goto out_unlock;
+ if (s->flags & SLAB_STORE_USER)
+ set_tracking(s, object, 1);
+ }
+
+ prior = object[page->offset] = page->freelist;
+ page->freelist = object;
+ page->inuse--;
+
+ if (likely(PageActive(page) || (page->inuse && prior)))
+ goto out_unlock;
+
+ if (!prior) {
+ /*
+ * The slab was full before. It will have one free
+ * object now. So move to the partial list.
+ */
+ add_partial(s, page);
+ goto out_unlock;
+ }
+
+ /*
+ * All object have been freed.
+ */
+ remove_partial(s, page);
+ slab_unlock(page);
+single_object_slab:
+ discard_slab(s, page);
+ local_irq_restore(flags);
+ return;
+
+out_unlock:
+ slab_unlock(page);
+ local_irq_restore(flags);
```

## [PATCH] SLUB The unqueued slab allocator V3

```
+}
+EXPORT_SYMBOL(kmem_cache_free);
+
+/* Figure out on which slab object the object resides */
+static __always_inline struct page *get_object_page(const void *x)
+{
+ struct page *page = virt_to_page(x);
+
+ if (unlikely(PageCompound(page)))
+ page = page->first_page;
+
+ if (!PageSlab(page))
+ return NULL;
+
+ return page;
+}
+
+/*
+ * kmem_cache_open produces objects aligned at "size" and the first object
+ * is placed at offset 0 in the slab (We have no metainformation on the
+ * slab, all slabs are in essence "off slab").
+ *
+ * In order to get the desired alignment one just needs to align the
+ * size.
+ *
+ * Notice that the allocation order determines the sizes of the per cpu
+ * caches. Each processor has always one slab available for allocations.
+ * Increasing the allocation order reduces the number of times that slabs
+ * must be moved on and off the partial lists and therefore may influence
+ * locking overhead.
+ *
+ * The offset is used to relocate the free list link in each object. It is
+ * therefore possible to move the free list link behind the object. This
+ * is necessary for RCU to work properly and also useful for debugging.
+ *
+ * No freelists are necessary if there is only one element per slab.
+ */
+
+/*
+ * Minimum order of slab pages. This influences locking overhead and slab
+ * fragmentation. A higher order reduces the number of partial slabs
+ * and increases the number of allocations possible without having to
+ * take the list_lock.
+ */
+static int slub_min_order = 0;
+
+/*
+ * Merge control. If this is set then no merging of slab caches will occur.
+ */
+static int slub_nomerge = 0;
+
```

## [PATCH] SLUB The unqueued slab allocator V3

```
+/*
+ * Debug settings:
+ */
+static int slub_debug = 0;
+
+static char *slub_debug_slabs = NULL;
+
+static int calculate_order(int size)
+{
+ int order;
+ int rem;
+
+ if ((size & (size - 1)) == 0) {
+ /*
+ * We can use the page allocator if the requested size
+ * is compatible with the page sizes supported.
+ */
+ int order = fls(size) - 1 - PAGE_SHIFT;
+
+ if (order >= 0)
+ return order;
+ }
+
+ for (order = max(slub_min_order, fls(size - 1) - PAGE_SHIFT);
+ order < MAX_ORDER; order++) {
+ unsigned long slab_size = PAGE_SIZE << order;
+
+ if (slab_size < size)
+ continue;
+
+ rem = slab_size % size;
+
+ if (rem * 8 <= PAGE_SIZE << order)
+ break;
+ }
+
+ if (order >= MAX_ORDER)
+ return -E2BIG;
+ return order;
+}
+
+static unsigned long calculate_alignment(unsigned long flags,
+ unsigned long align)
+{
+ if (flags & (SLAB_MUST_HWCACHE_ALIGN|SLAB_HWCACHE_ALIGN))
+ return L1_CACHE_BYTES;
+
+ if (align < ARCH_SLAB_MINALIGN)
+ return ARCH_SLAB_MINALIGN;
+
+ return ALIGN(align, sizeof(void *));
+}
```

## [PATCH] SLUB The unqueued slab allocator V3

```
+}
+
+void free_kmem_cache_nodes(struct kmem_cache *s)
+{
+#ifdef CONFIG_NUMA
+ int node;
+
+ for_each_online_node(node) {
+ struct kmem_cache_node *n = s->node[node];
+ if (n && n != &s->local_node)
+ kfree(n);
+ s->node[node] = NULL;
+ }
+#endif
+}
+
+static void init_kmem_cache_node(struct kmem_cache_node *n)
+{
+ memset(n, 0, sizeof(struct kmem_cache_node));
+ atomic_long_set(&n->nr_slabs, 0);
+ spin_lock_init(&n->list_lock);
+ INIT_LIST_HEAD(&n->partial);
+}
+
+int init_kmem_cache_nodes(struct kmem_cache *s, gfp_t gfpflags)
+{
+#ifdef CONFIG_NUMA
+ int node;
+ int local_node;
+
+ if (slab_state == UP)
+ local_node = page_to_nid(virt_to_page(s));
+ else
+ local_node = 0;
+
+ for_each_online_node(node) {
+ struct kmem_cache_node *n;
+
+ if (local_node == node)
+ n = &s->local_node;
+ else
+ if (slab_state == DOWN) {
+ /*
+ * No kmalloc_node yet so do it by hand.
+ * We know that this is the first slab on the
+ * node for this slabcache. There are no concurrent
+ * accesses possible. Which simplifies things.
+ */
+ unsigned long flags;
+ struct page *page;
+

```

## [PATCH] SLUB The unqueued slab allocator V3

```
+ local_irq_save(flags);
+ page = new_slab(s, gfpflags, node);
+
+ BUG_ON(!page);
+ n = page->freelist;
+ page->freelist = *(void **)page->freelist;
+ page->inuse++;
+ local_irq_restore(flags);
+ } else
+ n = kmemalloc_node(sizeof(struct kmem_cache_node),
+ gfpflags, node);
+
+ if (!n) {
+ free_kmem_cache_nodes(s);
+ return 0;
+ }
+
+ s->node[node] = n;
+ init_kmem_cache_node(n);
+
+ if (slab_state == DOWN)
+ atomic_long_inc(&n->nr_slabs);
+ }
+ #else
+ init_kmem_cache_node(&s->local_node);
+ #endif
+ return 1;
+ }
+
+ int kmem_cache_open(struct kmem_cache *s, gfp_t gfpflags,
+ const char *name, size_t size,
+ size_t align, unsigned long flags,
+ void (*ctor)(void *, struct kmem_cache *, unsigned long),
+ void (*dtor)(void *, struct kmem_cache *, unsigned long))
+ {
+ int tentative_size;
+ BUG_ON(flags & SLUB_UNIMPLEMENTED);
+
+ memset(s, 0, kmem_size);
+
+ /*
+  * Enable debugging if selected on the kernel commandline.
+  */
+ if (slub_debug &&
+ (!slub_debug_slabs ||
+ strncmp(slub_debug_slabs, name, strlen(slub_debug_slabs)) == 0))
+ flags |= slub_debug;
+
+ if ((flags & SLAB_POISON) && ((flags & SLAB_DESTROY_BY_RCU) ||
+ ctor || dtor)) {
+ if (!(slub_debug & SLAB_POISON))
```

## [PATCH] SLUB The unqueued slab allocator V3

```
+ printk(KERN_WARNING "SLUB %s: Clearing SLAB_POISON "  
+ "because de/constructor exists.\n",  
+ s->name);  
+ flags &= ~SLAB_POISON;  
+ }  
+  
+ tentative_size = ALIGN(size, calculate_alignment(align, flags));  
+  
+ /*  
+ * Single object slabs are passed through to the page allocator  
+ * and therefore the checks we can do are limited.  
+ */  
+ if (size * 2 >= (PAGE_SIZE << calculate_order(tentative_size)))  
+ flags &= ~(SLAB_RED_ZONE | SLAB_DEBUG_FREE | \  
+ SLAB_STORE_USER | SLAB_POISON);  
+  
+ s->name = name;  
+ s->ctor = ctor;  
+ s->dtor = dtor;  
+ s->objsize = size;  
+ s->flags = flags;  
+  
+ size = ALIGN(size, sizeof(void *));  
+  
+ /*  
+ * If we redzone then check if we have space through above  
+ * alignment. If not then add an additional word, so  
+ * that we have a guard value to check for overwrites.  
+ */  
+ if ((s->flags & SLAB_RED_ZONE) && size == s->objsize)  
+ size += sizeof(void *);  
+  
+ s->inuse = size;  
+  
+ if (size * 2 < (PAGE_SIZE << calculate_order(size)) &&  
+ ((flags & (SLAB_DESTROY_BY_RCU | SLAB_POISON)) ||  
+ ctor || dtor)) {  
+ /*  
+ * Relocate free pointer after the object if it is not  
+ * permitted to overwrite the first word of the object on  
+ * kmem_cache_free.  
+ *  
+ * This is the case if we do RCU, have a constructor or  
+ * destructor.  
+ */  
+ s->offset = size / sizeof(void *);  
+ size += sizeof(void *);  
+ }  
+  
+ if (flags & SLAB_STORE_USER)  
+ size += 2 * sizeof(void *);
```

## [PATCH] SLUB The unqueued slab allocator V3

```
+
+ align = calculate_alignment(flags, align);
+
+ size = ALIGN(size, align);
+ s->size = size;
+
+
+ s->order = calculate_order(size);
+ if (s->order < 0)
+ goto error;
+
+
+ s->objects = (PAGE_SIZE << s->order) / size;
+ if (!s->objects || s->objects > 65535)
+ goto error;
+
+
+ atomic_set(&s->refcount, 1);
+
+
+ #ifdef CONFIG_SMP
+ mutex_init(&s->flushing);
+ atomic_set(&s->cpu_slabs, 0);
+ INIT_DELAYED_WORK(&s->flush, flusher);
+ #endif
+ if (init_kmem_cache_nodes(s, gfpflags)) {
+ register_slab(s);
+ return 1;
+ }
+ error:
+ if (flags & SLAB_PANIC)
+ panic("Cannot create slab %s size=%lu realsize=%u "
+ "order=%u offset=%u flags=%lx\n",
+ s->name, (unsigned long)size, s->size, s->order,
+ s->offset, flags);
+ return 0;
+ }
+ EXPORT_SYMBOL(kmem_cache_open);
+
+ /*
+ * Check if a given pointer is valid
+ */
+ int kmem_ptr_validate(struct kmem_cache *s, const void *object)
+ {
+ struct page * page;
+ void *addr;
+
+ page = get_object_page(object);
+
+ if (!page || s != page->slab)
+ /* No slab or wrong slab */
+ return 0;
+
+ addr = page_address(page);
+ if (object < addr || object >= addr + s->objects * s->size)
```

## [PATCH] SLUB The unqueued slab allocator V3

```
+ /* Out of bounds */
+ return 0;
+
+ if ((object - addr) & s->size)
+ /* Improperly aligned */
+ return 0;
+
+ /*
+ * We could also check here if the object is on the slabs freelist.
+ * But this would be too expensive and it seems that the main
+ * purpose of kmem_ptr_valid is to check if the object belongs
+ * to a certain slab.
+ */
+ return 1;
+}
+EXPORT_SYMBOL(kmem_ptr_validate);
+
+/*
+ * Determine the size of a slab object
+ */
+unsigned int kmem_cache_size(struct kmem_cache *s)
+{
+ return s->objsize;
+}
+EXPORT_SYMBOL(kmem_cache_size);
+
+const char *kmem_cache_name(struct kmem_cache *s)
+{
+ return s->name;
+}
+EXPORT_SYMBOL(kmem_cache_name);
+
+static int free_list(struct kmem_cache *s, struct kmem_cache_node *n,
+ struct list_head *list)
+{
+ int slabs_inuse = 0;
+ unsigned long flags;
+ struct page *page, *h;
+
+ spin_lock_irqsave(&n->list_lock, flags);
+ list_for_each_entry_safe(page, h, list, lru)
+ if (!page
```