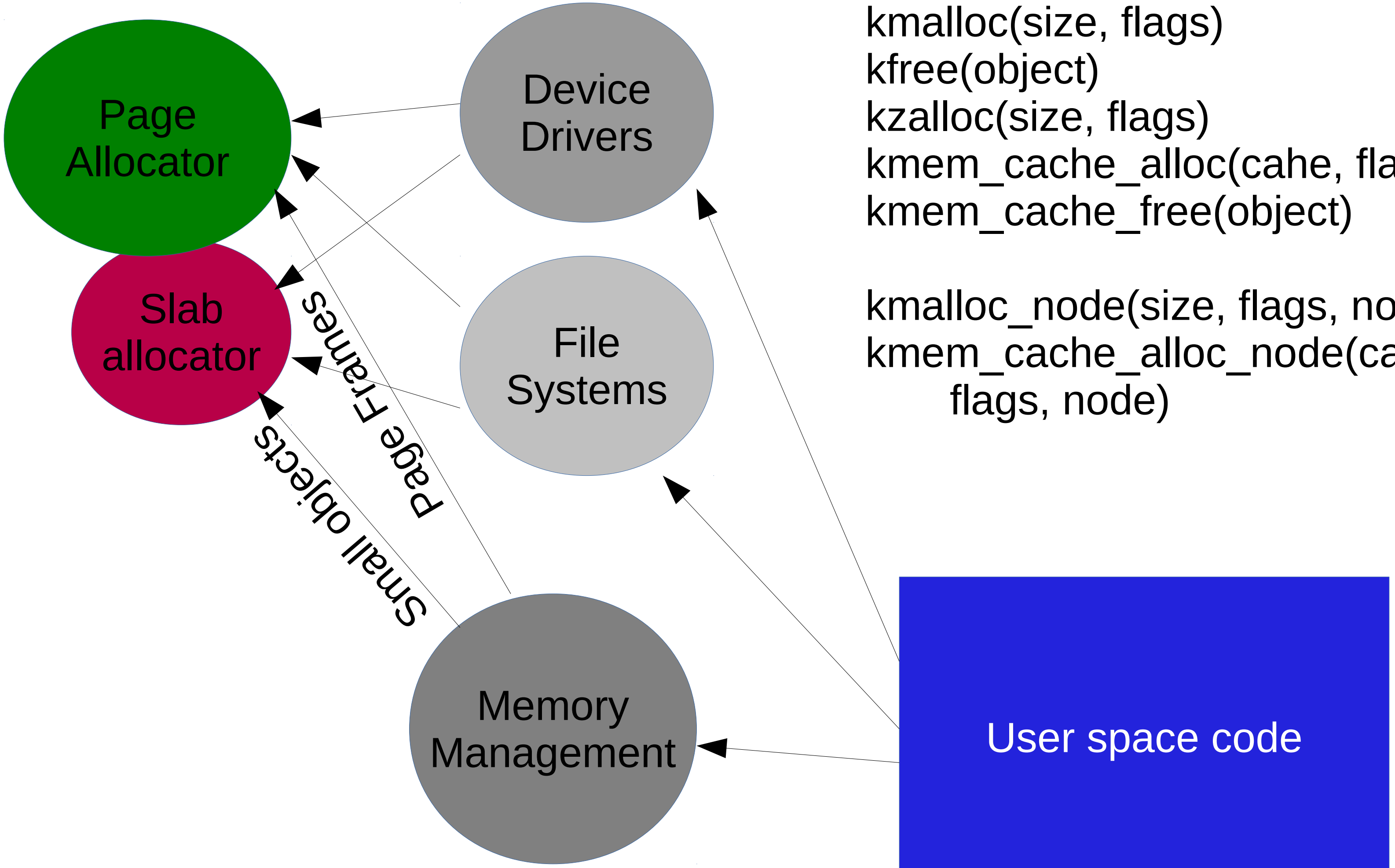
Slab allocators in the Linux Kernel: SLAB, SLOB, SLUB

Christoph Lameter,
LCA 2015 Auckland/New Zealand
(Revision Jan 15, 2015)

The Role of the Slab allocator in Linux

- PAGE_SIZE (4k) basic allocation unit via page allocator.
- Allows fractional allocation. Frequently needed for small objects that the kernel allocates f.e. for network descriptors.
- Slab allocation is very performance sensitive.
- Caching.
- All other subsystems need the services of the slab allocators.
- Terminology: SLAB is one of the slab allocator.
- A SLAB could be a page frame or a slab cache as a whole. It's confusing. Yes.

System Components around Slab Allocators



kmalloc(size, flags)
kfree(object)
kzalloc(size, flags)
kmem_cache_alloc(cahe, flags)
kmem_cache_free(object)

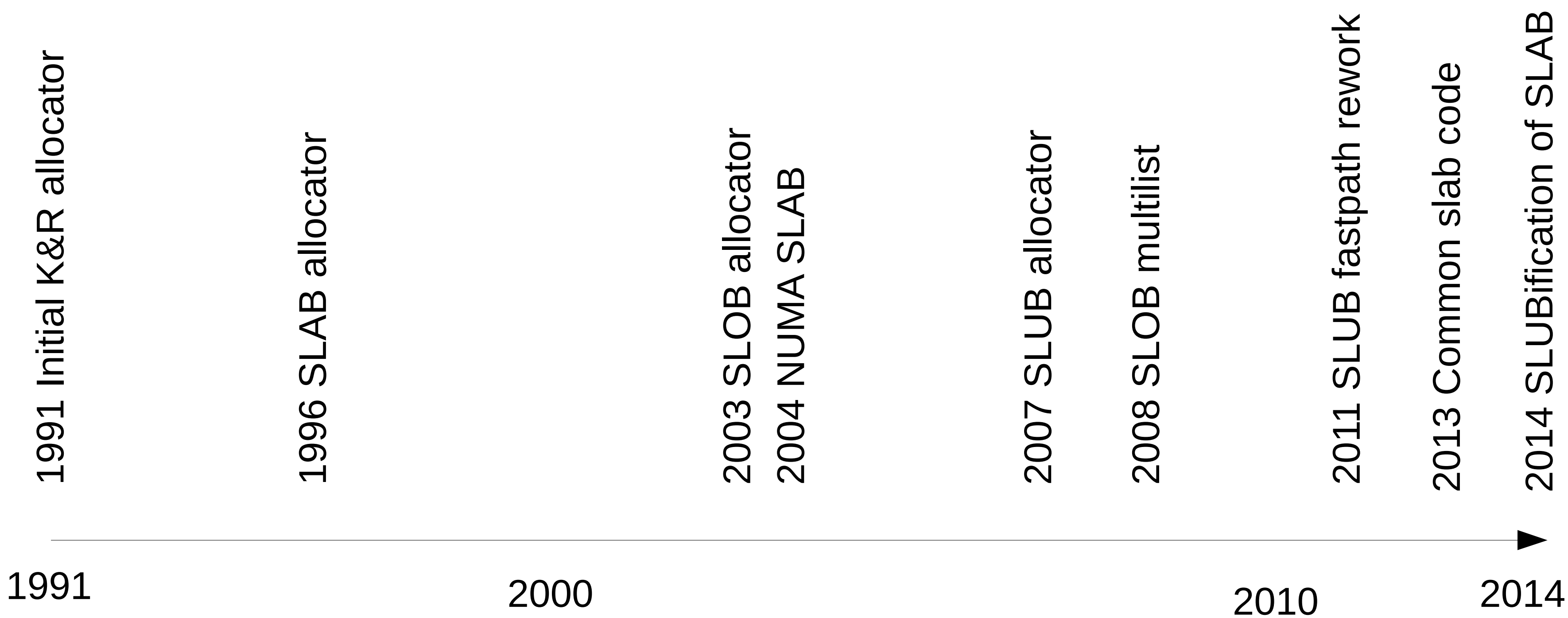
kmalloc_node(size, flags, node)
kmem_cache_alloc_node(cache,
    flags, node)

Slab allocators available

- SLOB: K&R allocator (1991-1999)
- SLAB: Solaris type allocator (1999-2008)
- SLUB: Unqueued allocator (2008-today)

- Design philosophies
  - SLOB: As compact as possible
  - SLAB: As cache friendly as possible. Benchmark friendly.
  - SLUB: Simple and instruction cost counts. Superior Debugging. Defragmentation. Execution time friendly.

Time line: Slab subsystem development



Maintainers

- Manfred Spraul <SLAB Retired>
- Matt Mackall <SLOB Retired>
- Pekka Enberg
- Christoph Lameter <SLUB, SLAB NUMA>
- David Rientjes
- Joonsoo Kim

Major Contributors

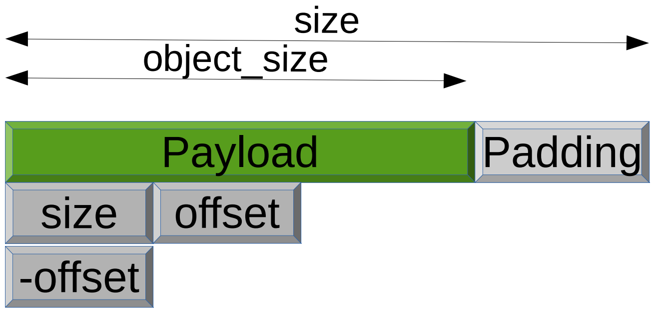
- Alokk N Kataria SLAB NUMA code
- Shobhit Dayal SLAB NUMA architecture
- Glauber Costa Cgroups support
- Nick Piggin SLOB NUMA support and performance optimizations. Multiple alternative out of tree implementations for SLUB.

Basic structures of SLOB

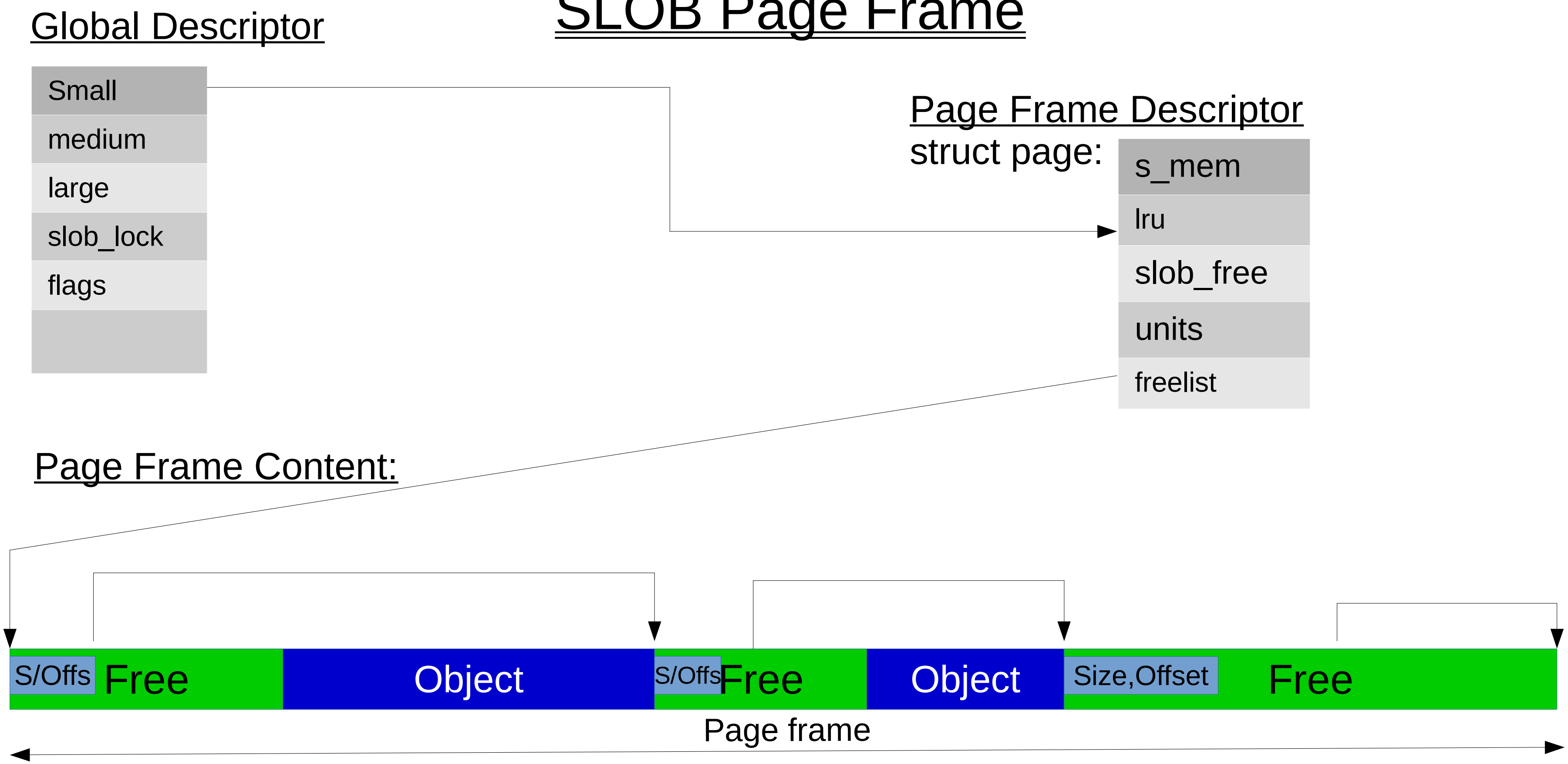
- K&R allocator: Simply manages list of free objects within the space of the free objects.
- Allocation requires traversing the list to find an object of sufficient size. If nothing is found the page allocator is used to increase the size of the heap.
- Rapid fragmentation of memory.
- Optimization: Multiple list of free objects according to size reducing fragmentation.

SLOB object format

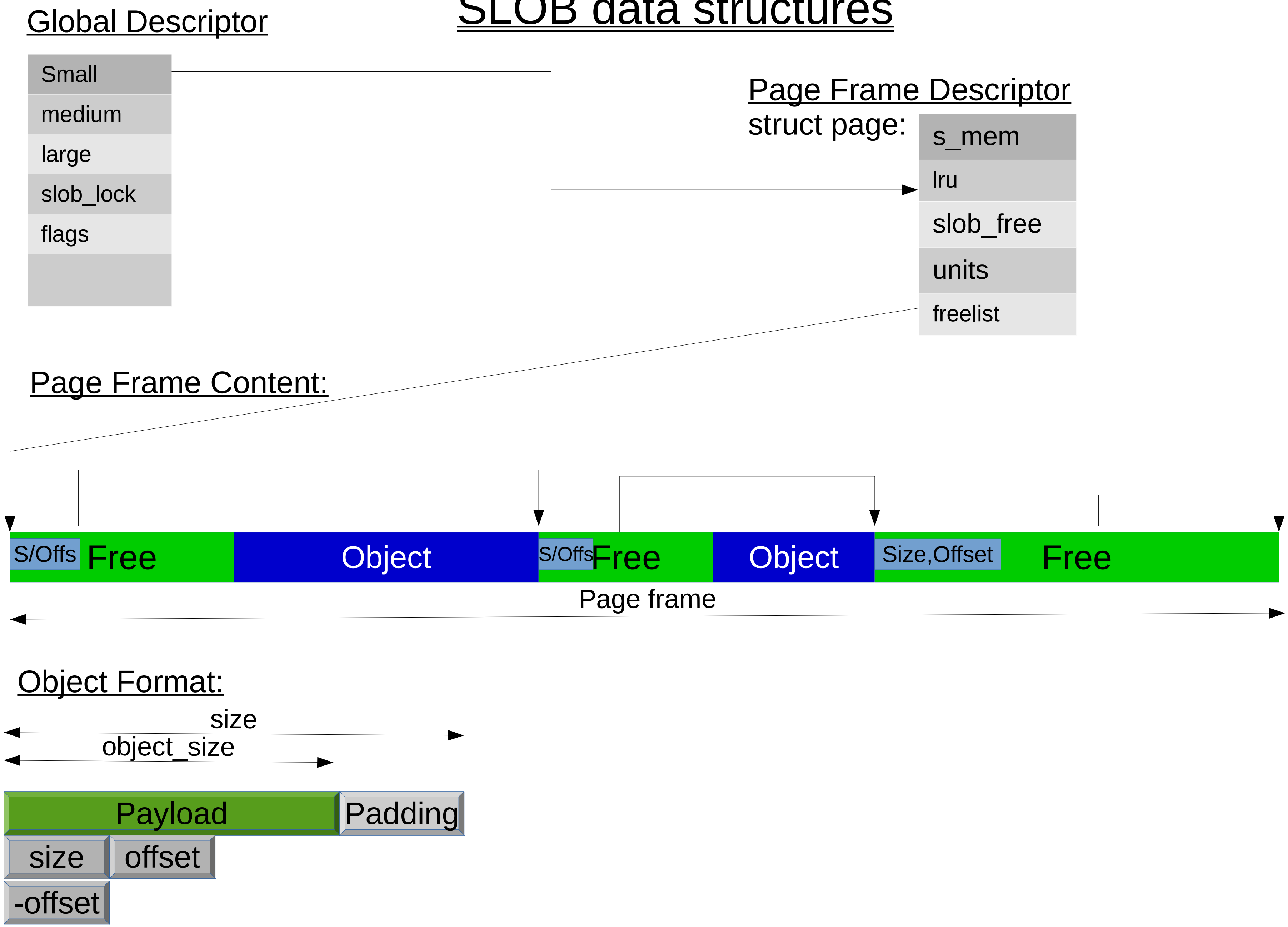
Object Format:



SLOB Page Frame



SLOB data structures

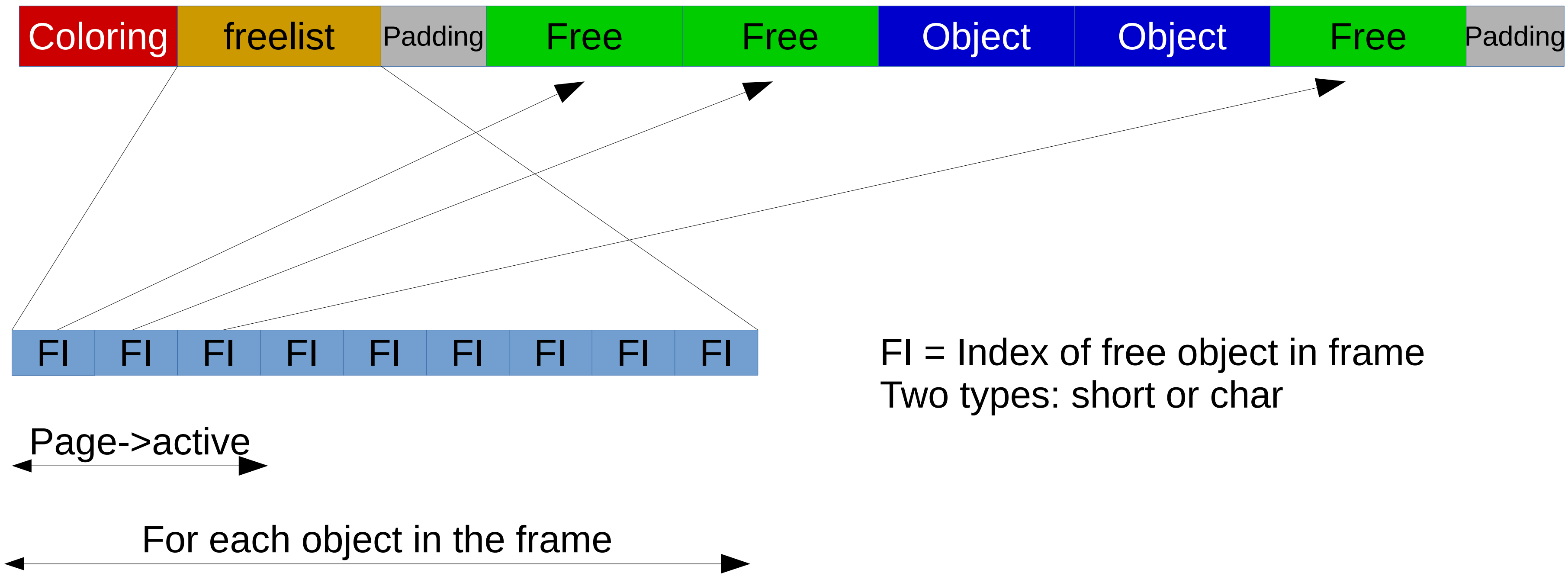


SLAB memory management

- Queues to track cache hotness
- Queues per cpu and per node
- Queues for each remote node (alien caches)
- Complex data structures that are described in the following two slides.
- Object based memory policies and interleaving.
- Exponential growth of caches *nodes * nr_cpus*. Large systems have huge amount of memory trapped in caches.
- Cold object expiration: Every processor has to scan its queues of every slab cache every 2 seconds.

SLAB per frame freelist management

Page Frame Content:

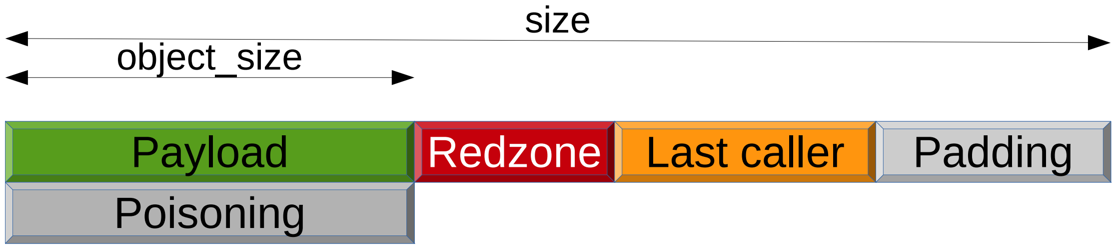


FI = Index of free object in frame
Two types: short or char

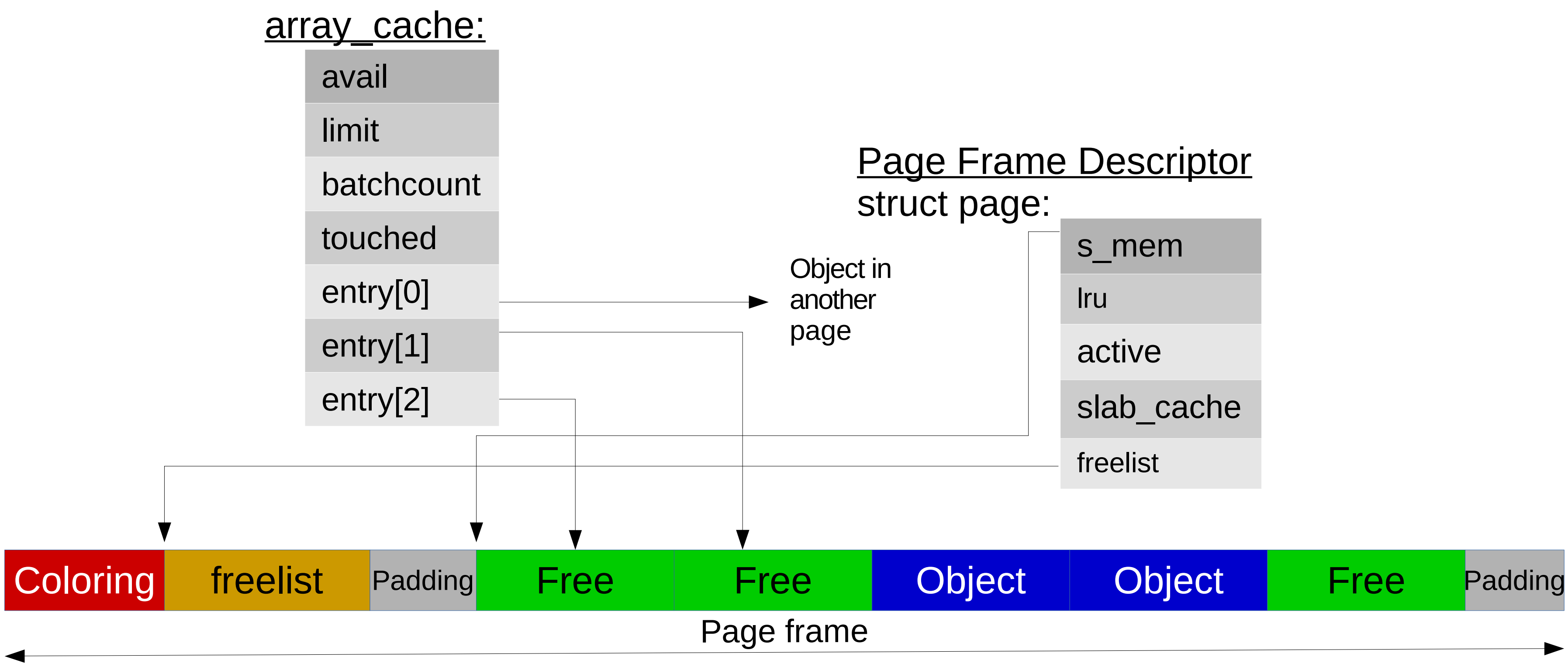
Multiple requests for free objects can be satisfied from the same cacheline without touching the object contents.

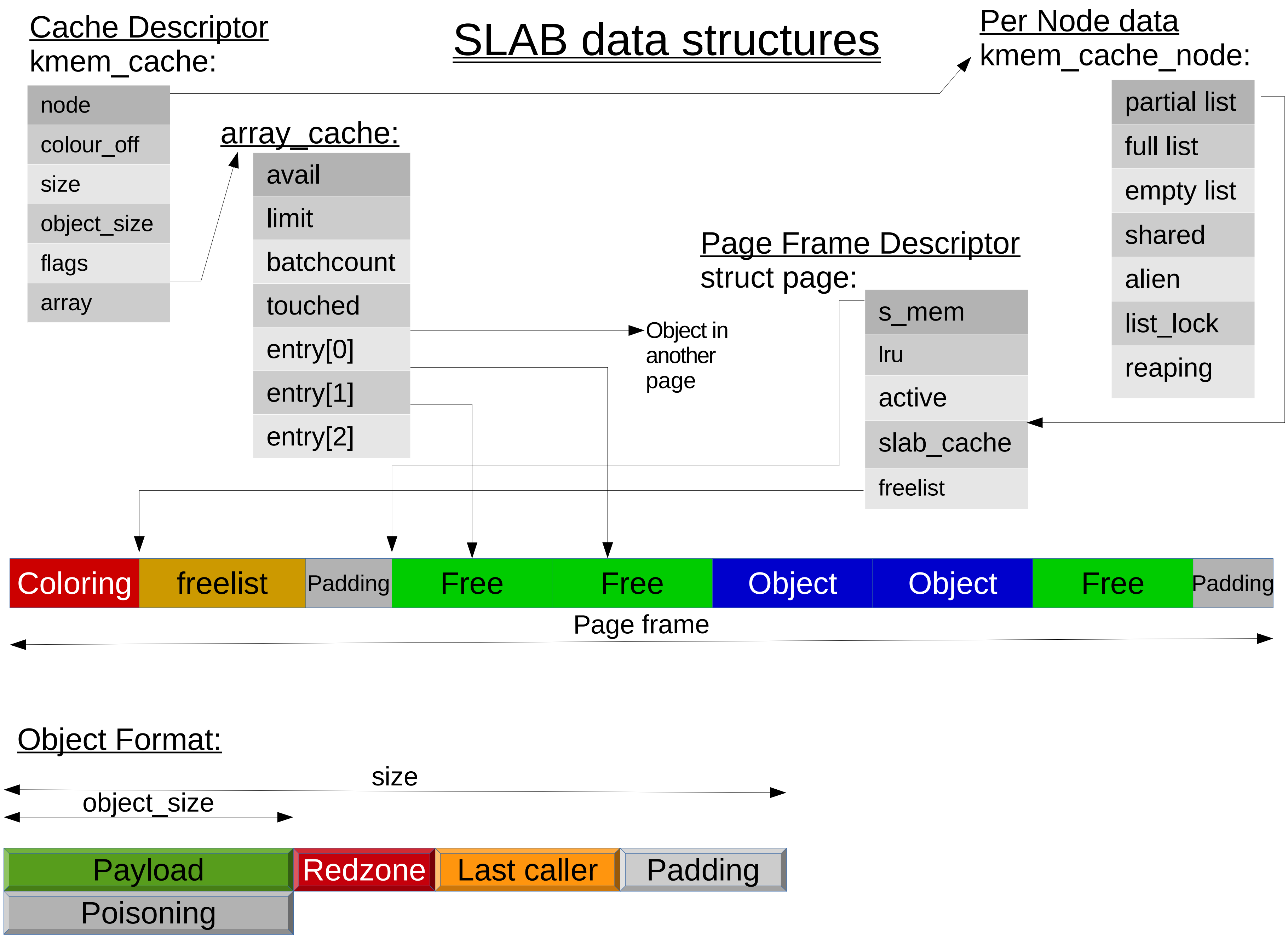
SLAB object format

Object Format:



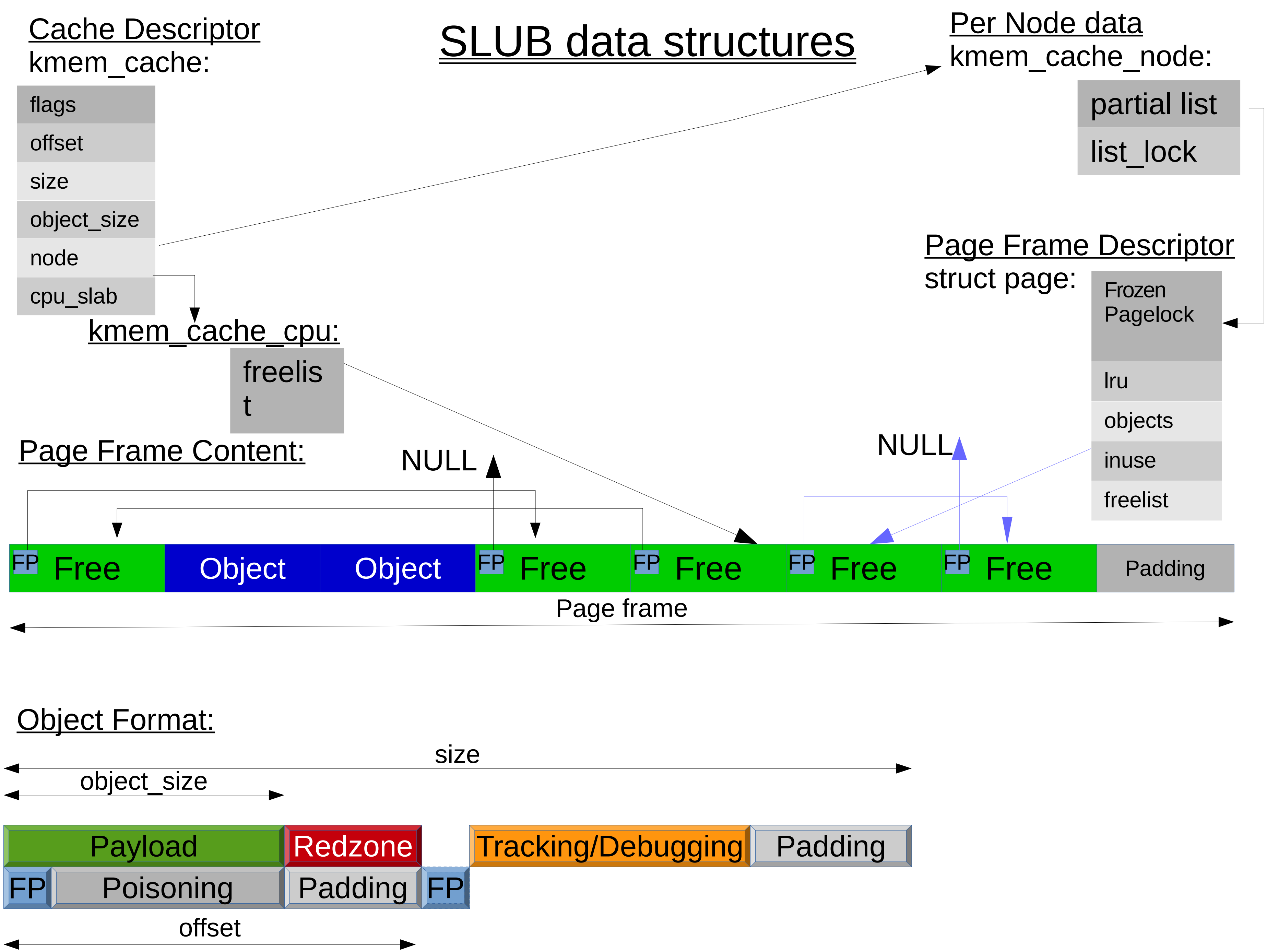
SLAB Page Frame




SLAB data structures
Cache Descriptor
kmem_cache:
node
colour_off
size
object_size
flags
array
array_cache:
avail
limit
batchcount
touched
entry[0]
entry[1]
entry[2]
Object in another page
Page Frame Descriptor
struct page:
s_mem
lru
active
slab_cache
freelist
Per Node data
kmem_cache_node:
partial list
full list
empty list
shared
alien
list_lock
reaping
Coloring
freelist
Padding
Free
Free
Object
Object
Free
Padding
Page frame
Object Format:
size
object_size
Payload
Redzone
Last caller
Padding
Poisoning

SLUB memory layout

- Enough of the queueing. An “Unqueued” allocator.
- “Queue” for a single slab page. Pages associated with per cpu. Increased locality.
- Per cpu partials
- Fast paths using this_cpu_ops and per cpu data.
- Page based policies and interleave.
- Defragmentation functionality on multiple levels.
- Current default slab allocator.



SLUB data structures
Cache Descriptor
kmem_cache:
flags
offset
size
object_size
node
cpu_slab
Per Node data
kmem_cache_node:
partial list
list_lock
Page Frame Descriptor
struct page:
Frozen
Pagelock
lru
objects
inuse
freelist
kmem_cache_cpu:
freelis
t
Page Frame Content:
NULL
NULL
FP Free
Object
Object
FP Free
FP Free
FP Free
FP Free
Padding
Page frame
Object Format:
size
object_size
Payload
Redzone
Tracking/Debugging
Padding
FP
Poisoning
Padding
FP
offset

SLUB slabinfo tool

- Query status of slabs and objects
- Control anti-defrag and object reclaim
- Run verification passes over slab caches
- Tune slab caches
- Modify slab caches on the fly

Slabinfo Examples

- Usually must be compiled from kernel source tree: gcc -o slabinfo linux/tools/vm/slabinfo.c
- Slabinfo
- Slabinfo -T
- Slabinfo -s
- Slabinfo -v

slabinfo basic output

Name	Objects	Objsize	Space	Slabs/Part/Cpu	O/S	O	%Fr	%Ef	Flg
:at-0000040	41635	40	1.6M	403/10/9	102	0	2	98	*a
:t-0000024	7	24	4.0K	1/1/0	170	0	100	4	*
:t-0000032	3121	32	180.2K	30/27/14	128	0	61	55	*
:t-0002048	564	2048	1.4M	31/13/14	16	3	28	78	*
:t-0002112	384	2112	950.2K	29/12/0	15	3	41	85	*
:t-0004096	412	4096	1.9M	48/9/10	8	3	15	88	*
Acpi-State	51	80	4.0K	0/0/1	51	0	0	99	
anon_vma	8423	56	647.1K	98/40/60	64	0	25	72	
bdev_cache	34	816	262.1K	8/8/0	39	3	100	10	Aa
blkdev_queue	27	1896	131.0K	4/3/0	17	3	75	39	
blkdev_requests	168	376	65.5K	0/0/8	21	1	0	96	
Dentry	191961	192	37.4M	9113/0/28	21	0	0	98	a
ext4_inode_cache	163882	976	162.8M	4971/15/0	33	3	0	98	a
Taskstats	47	328	65.5K	8/8/0	24	1	100	23	
TCP	23	1760	131.0K	3/3/1	18	3	75	30	A
TCPv6	3	1920	65.5K	2/2/0	16	3	100	8	A
UDP	72	888	65.5K	0/0/2	36	3	0	97	A
UDPv6	60	1048	65.5K	0/0/2	30	3	0	95	A
vm_area_struct	20680	184	3.9M	922/30/31	22	0	3	97	

Totals: slabinfo -T

Slabcache Totals

Slabcaches : 112	Aliases : 189->84	Active: 66
Memory used: 267.1M	# Loss : 8.5M	MRatio: 3%
# Objects : 708.5K	# PartObj: 10.2K	ORatio: 1%

Per Cache	Average	Min	Max	Total
#Objects	10.7K	1	192.0K	708.5K
#Slabs	350	1	9.1K	23.1K
#PartSlab	8	0	82	566
%PartSlab	34%	0%	100%	2%
PartObjs	1	0	2.0K	10.2K
% PartObj	25%	0%	100%	1%
Memory	4.0M	4.0K	162.8M	267.1M
Used	3.9M	32	159.9M	258.6M
Loss	128.8K	0	2.9M	8.5M

Per Object	Average	Min	Max
Memory	367	8	8.1K
User	365	8	8.1K
Loss	2	0	64

Aliasing: slabinfo -a

```
:at-0000040  <- ext4_extent_status btrfs_delayed_extent_op
:at-0000104  <- buffer_head sda2 ext4_prealloc_space
:at-0000144  <- btrfs_extent_map btrfs_path
:at-0000160  <- btrfs_delayed_ref_head btrfs_trans_handle
:t-0000016   <- dm_mpath_io kmalloc-16 ecryptfs_file_cache
:t-0000024   <- scsi_data_buffer numa_policy
:t-0000032   <- kmalloc-32 dnotify_struct sd_ext_cdb ecryptfs_dentry_info_cache pte_list_desc
:t-0000040   <- khugepaged_mm_slot Acpi-Namespace dm_io ext4_system_zone
:t-0000048   <- ip_fib_alias Acpi-Parse ksm_mm_slot jbd2_inode nsproxy ksm_stable_node ftrace_event_field
shared_policy_node fasync_cache
:t-0000056   <- uhci_urb_priv fanotify_event_info ip_fib_trie
:t-0000064   <- dmaengine-unmap-2 secpath_cache kmalloc-64 io ksm_rmap_item fanotify_perm_event_info fs_cache
tcp_bind_bucket ecryptfs_key_sig_cache ecryptfs_global_auth_tok_cache fib6_nodes iommu_iova anon_vma_chain
iommu_devinfo
:t-0000256   <- skbuff_head_cache sgpool-8 pool_workqueue nf_conntrack_expect request_sock_TCPv6 request_sock_TCP
bio-0 filp biovec-16 kmalloc-256
:t-0000320   <- mnt_cache bio-1
:t-0000384   <- scsi_cmd_cache ip6_dst_cache i915_gem_object
:t-0000416   <- fuse_request dm_rq_target_io
:t-0000512   <- kmalloc-512 skbuff_fclone_cache sgpool-16
:t-0000640   <- kioctx dio files_cache
:t-0000832   <- ecryptfs_auth_tok_list_item task_xstate
:t-0000896   <- ecryptfs_sb_cache mm_struct UNIX RAW PING
:t-0001024   <- kmalloc-1024 sgpool-32 biovec-64
:t-0001088   <- signal_cache dmaengine-unmap-128 PINGv6 RAWv6
:t-0002048   <- sgpool-64 kmalloc-2048 biovec-128
:t-0002112   <- idr_layer_cache dmaengine-unmap-256
:t-0004096   <- ecryptfs_xattr_cache biovec-256 names_cache kmalloc-4096 sgpool-128 ecryptfs_headers
```

Enabling of runtime Debugging

- Debugging support is compiled in by default. A distro kernel has the ability to go into debug mode where meaningful information about memory corruption can be obtained.
- Activation via slub_debug kernel parameter or via the slabinfo tool. slub_debug can take some parameters

Letter	Purpose
F	Enable sanity check that may impact performance
P	Poisoning. Unused bytes and freed objects are overwritten with poisoning values. References to these areas will show specific bit patterns.
U	User tracking. Record stack traces on allocate and free
T	Trace. Log all activity on a certain slab cache
Z	Redzoning. Extra zones around objects that allow to detect writes beyond object boundaries.

Sample Error report in dmesg

```
=============================================
BUG kmalloc-128: Object already free
-----------------------------------------------------------------------------

INFO: Allocated in rt61pci_probe_hw+0x3e5/0x6e0 [rt61pci] age=340 cpu=0 pid=21
INFO: Freed in rt2x00lib_remove_hw+0x59/0x70 [rt2x00lib] age=0 cpu=0 pid=21
INFO: Slab 0xc13ac3e0 objects=23 used=10 fp=0xdd59f6e0 flags=0x400000c3
INFO: Object 0xdd59f6e0 @offset=1760 fp=0xdd59f790

Bytes b4 0xdd59f6d0:  15 00 00 00 b2 8a fb ff 5a 5a 5a 5a 5a 5a 5a 5a ....².ûÿZZZZZZZZ
  Object 0xdd59f6e0:  6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b kkkkkkkkkkkkkkkk
  Object 0xdd59f6f0:  6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b kkkkkkkkkkkkkkkk
  Object 0xdd59f700:  6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b kkkkkkkkkkkkkkkk
  Object 0xdd59f710:  6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b kkkkkkkkkkkkkkkk
  Object 0xdd59f720:  6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b kkkkkkkkkkkkkkkk
  Object 0xdd59f730:  6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b kkkkkkkkkkkkkkkk
  Object 0xdd59f740:  6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b kkkkkkkkkkkkkkkk
  Object 0xdd59f750:  6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b a5 kkkkkkkkkkkkkkk¥
 Redzone 0xdd59f760:  bb bb bb bb                                     »»»»
 Padding 0xdd59f788:  5a 5a 5a 5a 5a 5a 5a 5a                         ZZZZZZZZ
Pid: 21, comm: stage1 Not tainted 2.6.29.1-desktop-1.1mnb #1
Call Trace:
 [<c01abbb3>] print_trailer+0xd3/0x120
 [<c01abd37>] object_err+0x37/0x50
 [<c01acf57>] __slab_free+0xe7/0x2f0
 [<c026c9b0>] __pci_register_driver+0x40/0x80
 [<c03ac2fb>] ? mutex_lock+0xb/0x20
 [<c0105546>] syscall_call+0x7/0xb
FIX kmalloc-128: Object at 0xdd59f6e0 not freed
```

Comparing memory use

- SLOB most compact (unless frequent freeing and allocation occurs)
- SLAB queueing can get intensive memory use going. Grows exponentially by NUMA node.
- SLUB aliasing of slabs
- SLUB cache footprint optimizations
- Kvm instance memory use of allocators

Memory use after bootup of a desktop Linux system

Allocator	Reclaimable	Unreclaimable
SLOB*	~300KB +	
SLUB	29852 kB	32628 kB
SLAB	29028 kB	36532 kB

*SLOB does not support the slab statistics counters. 300Kb is the difference of “MemAvailable” after boot between SLUB and SLOB

Comparing performance

- SLOB is slow (atomics in fastpath, global locking)
- SLAB is fast for benchmarking
- SLUB is fast in terms of cycles used for the fastpath but may have issues with caching.
- SLUB is compensating for caching issues with an optimized fastpath that does not require interrupt disabling etc.
- Cache footprints are a main factor for performance these days. Benchmarking reserves most of the cache available for the slab operations which may be misleading.

Fastpath performance

Cycles	Alloc	Free	Alloc/Free	Alloc Concurrent	Free Concurrent
SLAB	66	73	102	232	984
SLUB	45	70	52	90	119
SLOB	183	173	172	3008	3037

Times in cycles on a Haswell 8 core desktop processor.
The lowest cycle count is taken from the test.

Hackbench comparison

Seconds	15 groups 50 filedesc 2000 messages 512 bytes
SLAB	4.92 4.87 4.85 4.98 4.85
SLUB	4.84 4.75 4.85 4.9 4.8
SLOB	N/A

Remote freeing

Cycles	Alloc all Free on one	Alloc one Free all
SLAB	650	761
SLUB	595	498
SLOB	2650	2013

Remote freeing is the freeing of an object that was allocated on a different Processor. Its cache cold and may have to be reused on the other processor. Remote freeing is a performance critical element and the reason that “alien” caches exist in SLAB. SLAB's alien caches exist for every node and every processor.

Future Roadmap

- Common slab framework (mm/slab_common.c)
- Move toward per object logic for Defragmentation and maybe to provide an infrastructure for generally movable objects (patchset done 2007-2009 maybe redo it)
- SLAB fastpath relying on this_cpu operations.
- SLUB fastpath cleanup. Remove preempt enable/disable for better CONFIG_PREEMPT performance [queued for 3.20].

Batch API

- Proposal posted in Dec.
  https://lkml.org/lkml/2014/12/18/329
- Freeing operation
  void kmem_cache_free_array
  (struct kmem_cache *, int nr, void **);
- Allocation
  int kmem_cache_alloc_array(struct
  kmem_cache *, gfp_t, int, void **, unsigned flags);
- Fallback implementation if the particular slab allocator does not provide the implementation to use the existing functions that do single object allocation.

Bulk Alloc Modes

- SLAB_ARRAY_ALLOC_LOCAL
 Allocation Objects that are already cached for allocation by the local processor. No locks will be taken. [Very fast but limited number of objects]
- SLAB_ARRAY_ALLOC_PARTIAL
 Allocate objects in slab pages that are not fully used on the local node [Preserve local objects and defrag friendly]
- SLAB_ARRAY_ALLOC_NEW
 Allocate new pages from the page allocator and use them to create lists of objects. [Fast allocation mode for really large bulk allocation].
- SLAB_ARRAT_ALLOC_FULL
 Fill up the array of pointers to the end even if there are not enough object available using the above methods.

Implementing Bulk alloc in SLUB

- Draft was posted in the discussion of the bulk alloc API. The key problem here is the freelist requiring object data access.
- Pages can be taken off the partial lists and the freelists are traversed to construct the object pointer array. There is only t need to take the node lock a single time for multiple partial slabs that may be available.
- Pages can be directly allocated from the page allocator avoiding the construction of the freelist in the first place.

Implementing Bulk alloc in SLAB

- Freelist can be traversed in a cache frienc way.
- There are already arrays of pointers in the per cpu, per node and alien queues.
- Pages can also be taken off the partial list and the pointer arrays can then be generated from the table of free objects.

SLUB Fastpath architecture

- Fastpaths are lockless and do not disable interrupts or preemption [too costly].
- Instead speculative operations are done and a single per cpu “atomic:” operation then is used to either commit or retry the operations using a this_cpu_cmpxchg_double.
- Cuts the number of cycles spent in fastpath down to half.

```
retry:
	c = this_cpu_ptr(s->cpu_slab);
	tid = c->tid;
	object = c->freelist;
	page = c->page;
	if (unlikely(!object || !node_match(page, node))) {
		object = __slab_alloc(s, gfpflags, node, addr, c);
		stat(s, ALLOC_SLOWPATH);
	} else {
		void *next_object = get_freepointer_safe(s, object);

		if (unlikely(!this_cpu_cmpxchg_double(
				s->cpu_slab->freelist, s->cpu_slab->tid,
				object, tid,
				next_object, next_tid(tid))))
			goto retry;
	}
```

Recent fastpath improvements in SLUB

- CONFIG_PREEMPT requires preempt_enable/disable in the fastpath which significantly increases allocator latencies.
- I proposed a rather complex approach but Joonsoo Kim found a simpler one. And that is going to be merged for the Linux 3.20.
- Major distros do not use CONFIG_PREEMPT. Mostly helps folks using RT kernels to keep allocator performance on par with non RT.

Conclusion

- Questions
- Suggestions
- New ideas