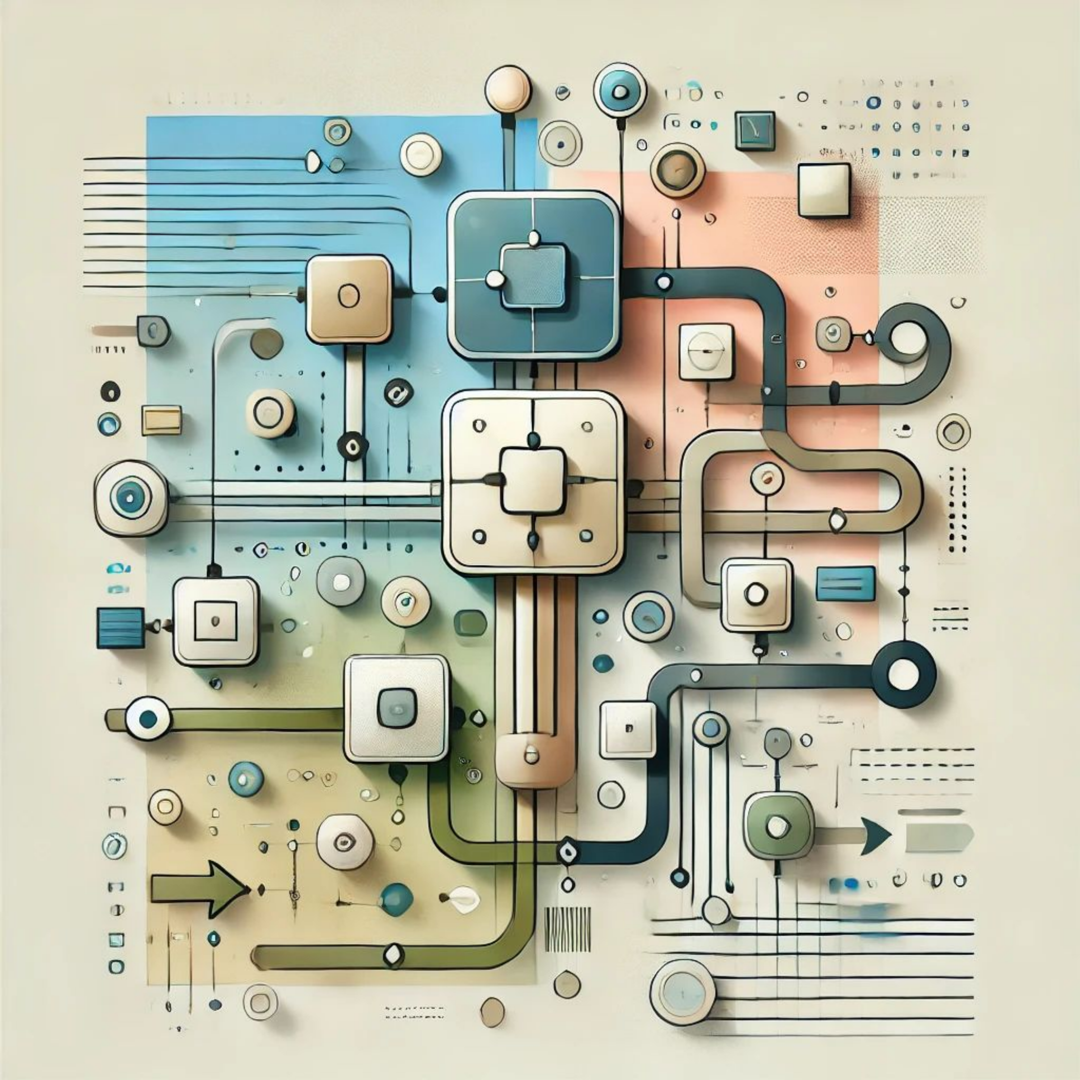


SLUB Internals for Exploit Developers

Andrey Konovalov, xairy.io

Linux Security Summit, Vienna
Sep 16th, 2024

Why this talk

- Few/no resources that connect SLUB internals with Slab shaping in exploits
  - Exploit write-ups usually only briefly cover Slab shaping for specific bug
  - Articles that focus on SLUB don't cover Slab shaping in exploits

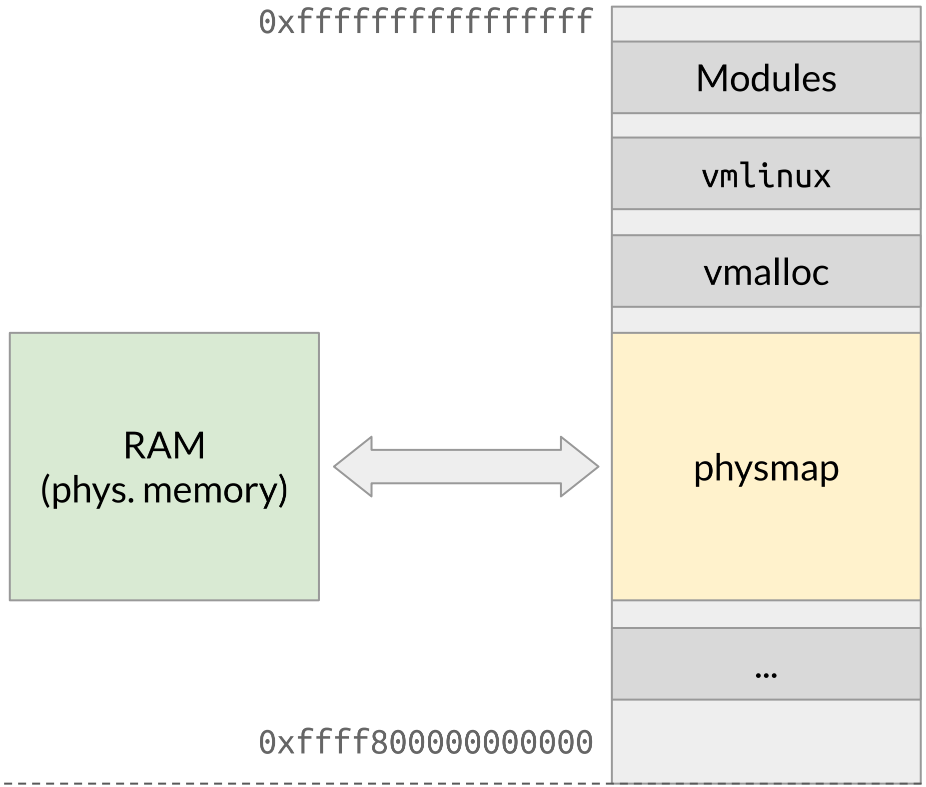
- This talk aims to fill the void and cover both:
  - Core SLUB internals: memory layout, allocation, and freeing
  - How Slab shaping works for basic memory corruption bugs based on knowledge of SLUB internals

Dynamic memory allocators in the Linux kernel

Link to slides

(Skipped in presentation mode)

Linux kernel virtual memory layout (x86-64)



Direct linear mapping of RAM
(except memory reserved by bootloader/BIOS)

[Kernel address space]

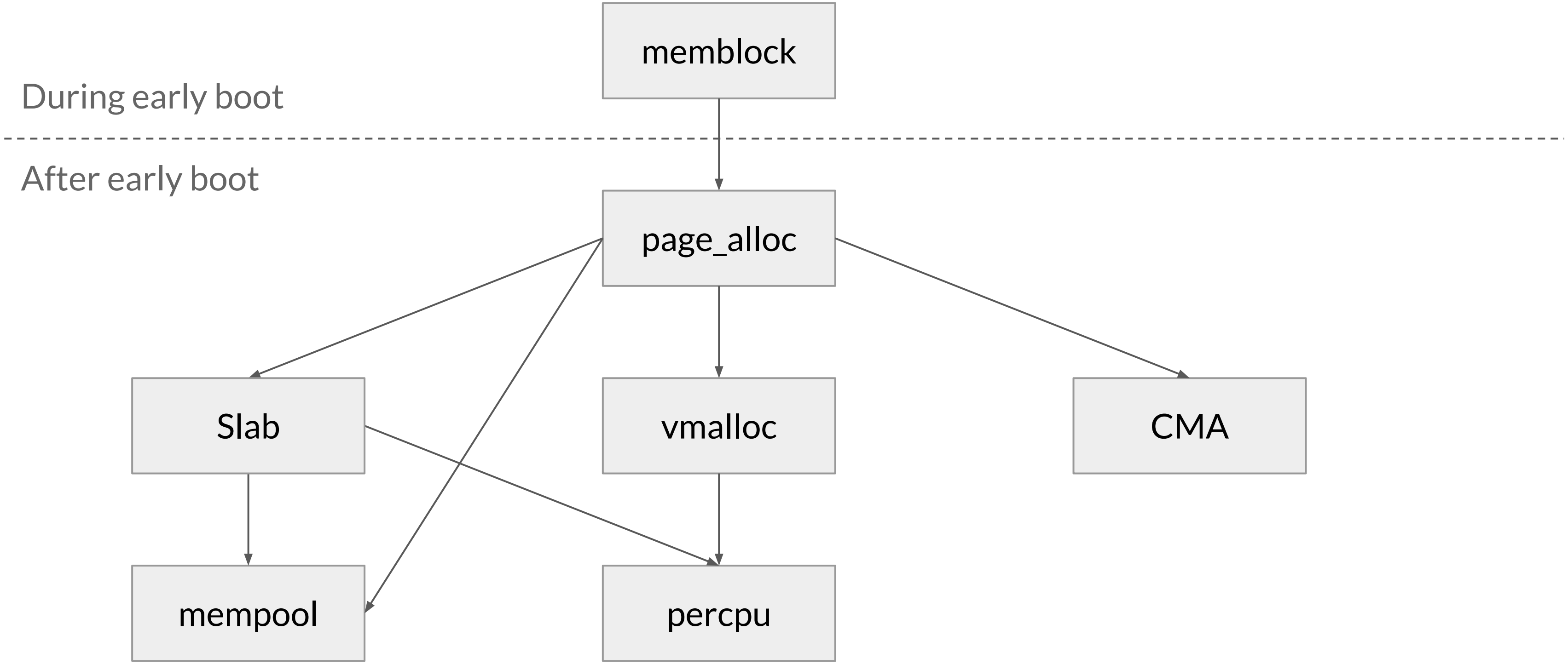
Dynamic memory allocators

- Non-reserved physmap memory passed to dynamic memory allocators

- Many of them:
  - During early boot: memblock
  - Physical memory: Slab, mempool, page_alloc, CMA
  - Virtual memory: vmalloc, percpu (backing memory is still physical)

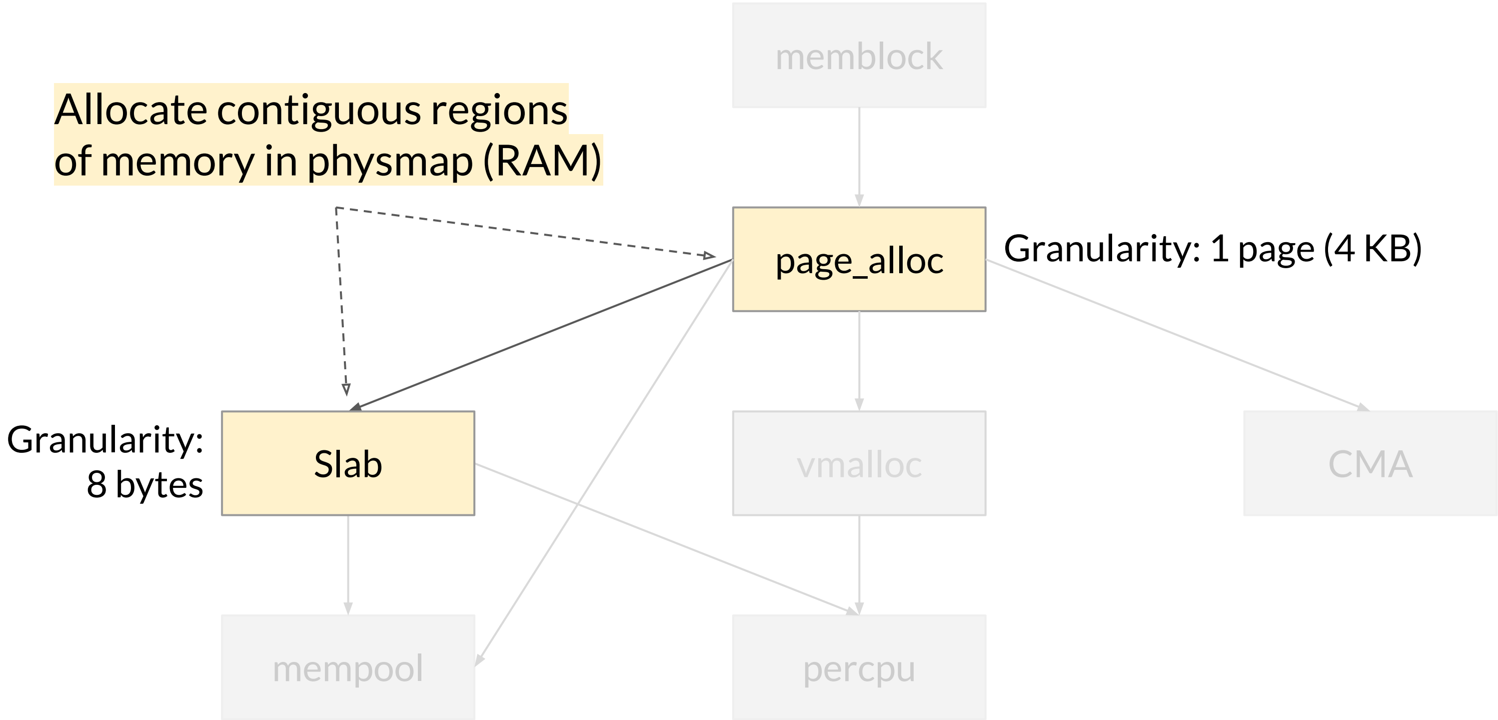
- Most often, memory corruptions affect Slab allocations
  - In write-ups, "heap" usually means Slab

Dynamic memory allocators hierarchy [partial]



Slab and page_alloc in hierarchy

Allocate contiguous regions of memory in physmap (RAM)

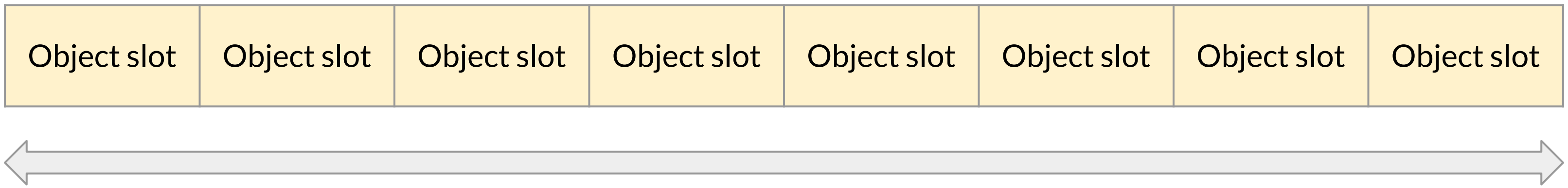


Slab allocator:
Core idea and API

(Partially skipped in presentation mode)

Slab allocator: slabs and objects

- Slab allows allocating small regions of physical memory called objects
- Slab stores objects in slabs — big regions intended for objects of the same size
    - Backing memory for slabs is allocated from page_alloc
- Slab allocates and frees objects from/to slots in these slabs



One slab with 8 object slots of the same size;
backing memory allocated from page_alloc

Slab allocator API

- On API side, objects are allocated from caches
  - Internally, caches allocate/free slabs to store objects

- Slab allocator provides two types of caches:
  1. Type-specific caches: each serves specific type of object with fixed size
  2. General-purpose (aka `kmalloc`) caches: used for generic allocations of varying sizes

Slab allocator API: type-specific caches

- Creating cache to allocate objects of size `size`:

```
struct kmem_cache *kmem_cache_create(const char *name, unsigned int size,
                                     unsigned int align, slab_flags_t flags,
                                     void (*ctor)(void *));
```

- Allocating and freeing objects from/to created cache:

```
void *kmem_cache_alloc(struct kmem_cache *s, gfp_t flags);
void kmem_cache_free(struct kmem_cache *s, void *objp);
```

kmem_cache_create/alloc/free usage example

```
cred_jar = kmem_cache_create("cred_jar", sizeof(struct cred), 0,
                SLAB_HWCACHE_ALIGN|SLAB_PANIC|SLAB_ACCOUNT, NULL);

struct cred *new = kmem_cache_alloc(cred_jar, GFP_KERNEL);

// Use new.

kmem_cache_free(cred_jar, new);
```

Slab allocator: `kmalloc` caches

- Slab allocator precreates caches for generic usage — `kmalloc` caches
- `kmalloc-8`, `kmalloc-16`, `kmalloc-32`, `kmalloc-64`, `kmalloc-96`, `kmalloc-128`, `kmalloc-192`, `kmalloc-256`, `kmalloc-512`, `kmalloc-1k`, `kmalloc-2k`, `kmalloc-4k`, `kmalloc-8k` on Ubuntu
- `sudo cat /proc/slabinfo` to see all caches

Slab allocator API: `kmalloc`

- Allocating and freeing objects from/to `kmalloc` caches:

```
void *kmalloc(size_t size, gfp_t flags);
void kfree(const void *objp);
```

- `kmalloc` uses smallest fitting `kmalloc` cache
  - Example: `kmalloc(100, GFP_KERNEL)` uses `kmalloc-128`

kmalloc/kfree usage example

```
void *data = kmalloc(142, GFP_KERNEL);

// Use data.

kfree(data);
```

Slab allocator variants

- Linux kernel (used to) provides 3 variants of Slab allocator
  - All follow the same API but have different implementations

- SLUB — Default allocator, used most frequently
  - Ubuntu and Android use SLUB
- SLAB — Reportedly faster than SLUB for certain workloads
  - Removed in 6.8 (all users moved to SLUB)
- SLOB — Has small memory footprint, used for embedded devices
  - Removed in 6.4 (`CONFIG_SLUB_TINY=y` to be used instead)

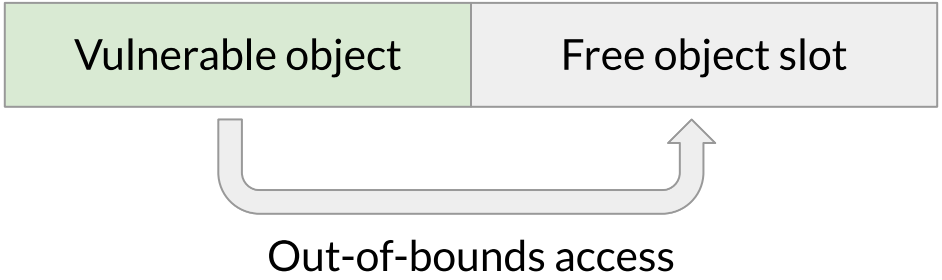
Slab bugs

Slab bugs

- Are typical bugs with dynamic memory:
    - Out-of-bounds (OOB) read or write
    - Use-after-free (UAF) read or write
    - Double-free and invalid-free

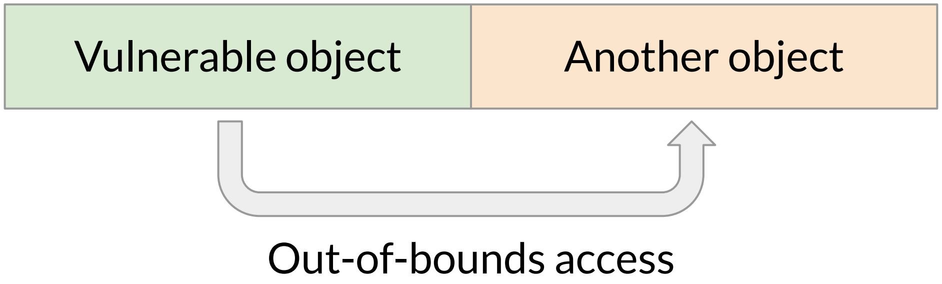
- Focus on OOB and UAF
    - Some notes on double-free at the end, invalid-free is out-of-scope

Out-of-bounds scenario example #1



- Vulnerable object — particular object affected by bug
- Free object slot — another slot in the same slab not occupied by any object

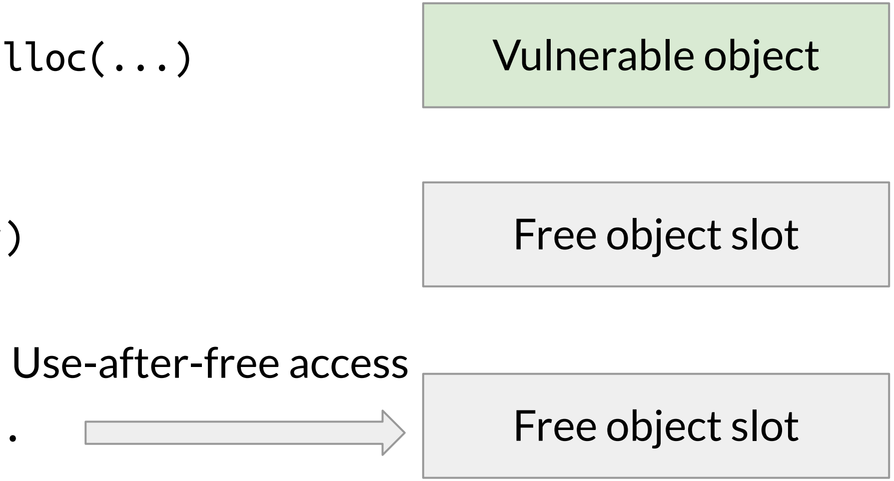
Out-of-bounds scenario example #2



- Vulnerable object — particular object affected by bug
- Another object — some object that follows vulnerable object in slab

Use-after-free scenario example #1

1. `vuln_ptr = kmalloc(...)`
2. `kfree(vuln_ptr)`
3. `*vuln_ptr = ...`



Use-after-free scenario example #2

1. `vuln_ptr = kmalloc(...)` — Vulnerable object

2. `kfree(vuln_ptr)` — Free object slot

3. Another object allocated in slot — Another object

4. `*vuln_ptr = ...` → Use-after-free access → Another object

Goal of Slab attack

- Use Slab bug to overwrite and/or leak either:
  - Metadata in free object slot
    - For example, to gain better Slab corruption primitive
    - Spoiler: possible but tricky
  - Another object
    - For example, leak `vmlinux` address to bypass KASLR
    - Or gain a better primitive (Arbitrary Address Execution or AARW) to escalate privileges
    - Spoiler: easy and thus the go-to approach

Required for attack [1/3]

1. Need kernel bug that leads to OOB or UAF access for vulnerable object
    - Will analyze a few different scenarios
    - Discussing specific bugs is out-of-scope

Required for attack [2/3]

1. Need kernel bug that leads to OOB or UAF access for vulnerable object

2. Need to choose target object to leak or overwrite
   - Many good objects, discussing specific ones is out-of-scope
   - Assume we chose one

Required for attack [3/3]

1. Need kernel bug that leads to OOB or UAF access for vulnerable object

2. Need to choose target object to leak or overwrite

3. Need to shape (aka groom or massage) Slab memory
   - For OOB: Put vulnerable and target object next to each other
   - For UAF: Put target object into slot of freed vulnerable object

- SLUB internals and shaping approaches are focus of the talk

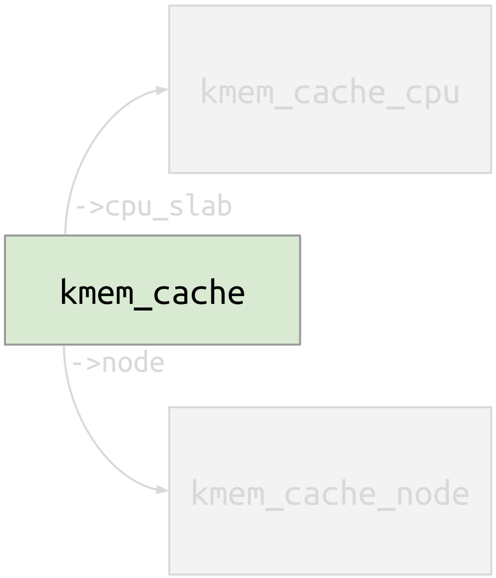
SLUB internals

Constraints

- Latest longterm kernel 6.6 (things might change in the future)
- Typical distro (Ubuntu) kernel config:
  - `CONFIG_SLUB_CPU_PARTIAL=y`
  - `# CONFIG_SLUB_DEBUG_ON is not set`
  - `# CONFIG_SLUB_TINY is not set`
  - ...
- Focus on basics: locking details and racy Slab shaping are out-of-scope
- Focus on single cache: cache merging and cross-cache are out-of-scope

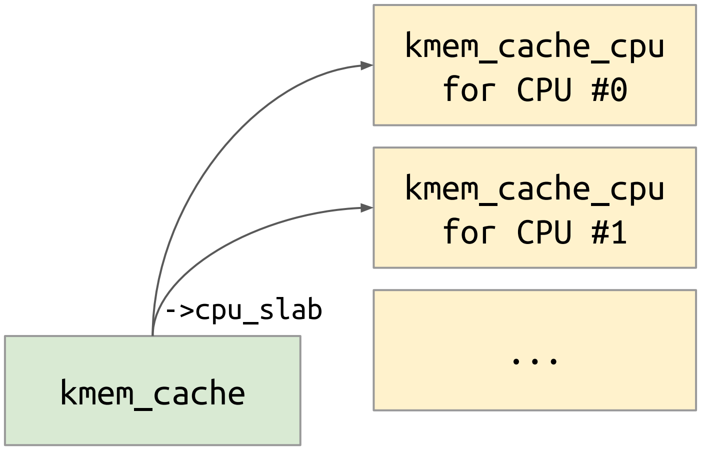
SLUB internals: `struct kmem_cache`

Each cache has kmem_cache structure




```
struct kmem_cache {
	// Per-CPU cache data:
	struct kmem_cache_cpu __percpu *cpu_slab;
	// Per-node cache data:
	struct kmem_cache_node *node[MAX_NUMNODES];
	...
	const char *name;          // Cache name
	slab_flags_t flags;        // Cache flags
	unsigned int object_size;  // Size of objects
	unsigned int offset;       // Freelist pointer offset
	unsigned long min_partial;
	unsigned int cpu_partial_slabs;
};
```

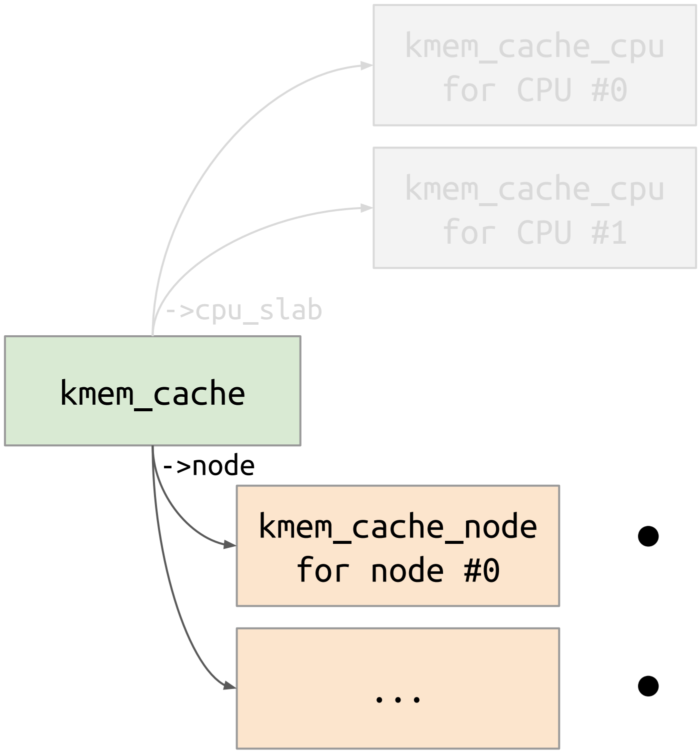
kmem_cache has per-CPU state kmem_cache_cpu




```
struct kmem_cache {
	// Per-CPU cache data:
	struct kmem_cache_cpu __percpu *cpu_slab;
	// Per-node cache data:
	struct kmem_cache_node *node[MAX_NUMNODES];
	...
};
```

- One instance of kmem_cache_cpu for each CPU (core)
- Assume we have only one CPU for now for simplicity

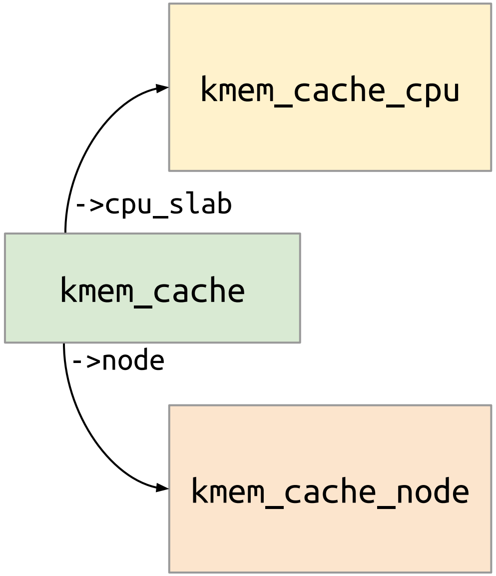
kmem_cache has per-node state kmem_cache_node




```
struct kmem_cache {
	// Per-CPU cache data:
	struct kmem_cache_cpu __percpu *cpu_slab;
	// Per-node cache data:
	struct kmem_cache_node *node[MAX_NUMNODES];
	...
};
```

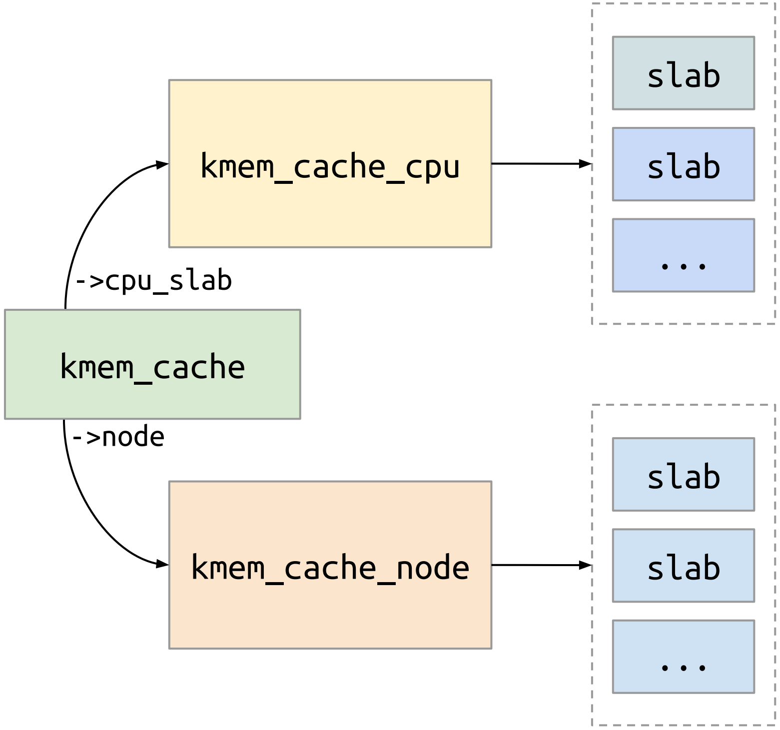
- One kmem_cache_node for each NUMA node
    - Non-Uniform Memory Access
- Assume one node (typical non-server system)

Assume one CPU and one NUMA node



- Assume one CPU for now
  - Effectively true during Slab shaping due to CPU pinning (explained later)
- Assume one NUMA node
  - Effectively true during Slab shaping: interacting with node slabs brings unreliability and typically avoided anyway
  - SLUB behaviour with multiple nodes is out-of-scope

kmem_cache_cpu/node contain pointers to slabs



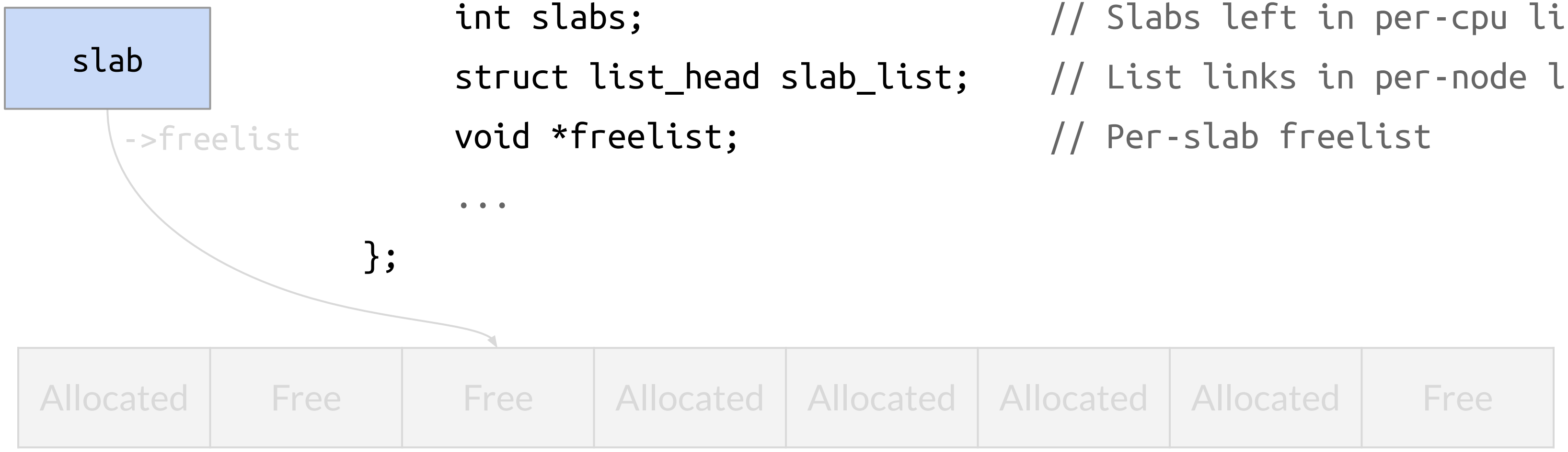

```
struct kmem_cache_cpu {
	struct slab *slab;    // Active slab
	struct slab *partial; // Partial slabs
	...
};

struct kmem_cache_node {
	struct list_head partial; // Slabs
	...
};
```

- Before studying these pointers, let's explore `struct slab`

SLUB internals: `struct slab` and its memory

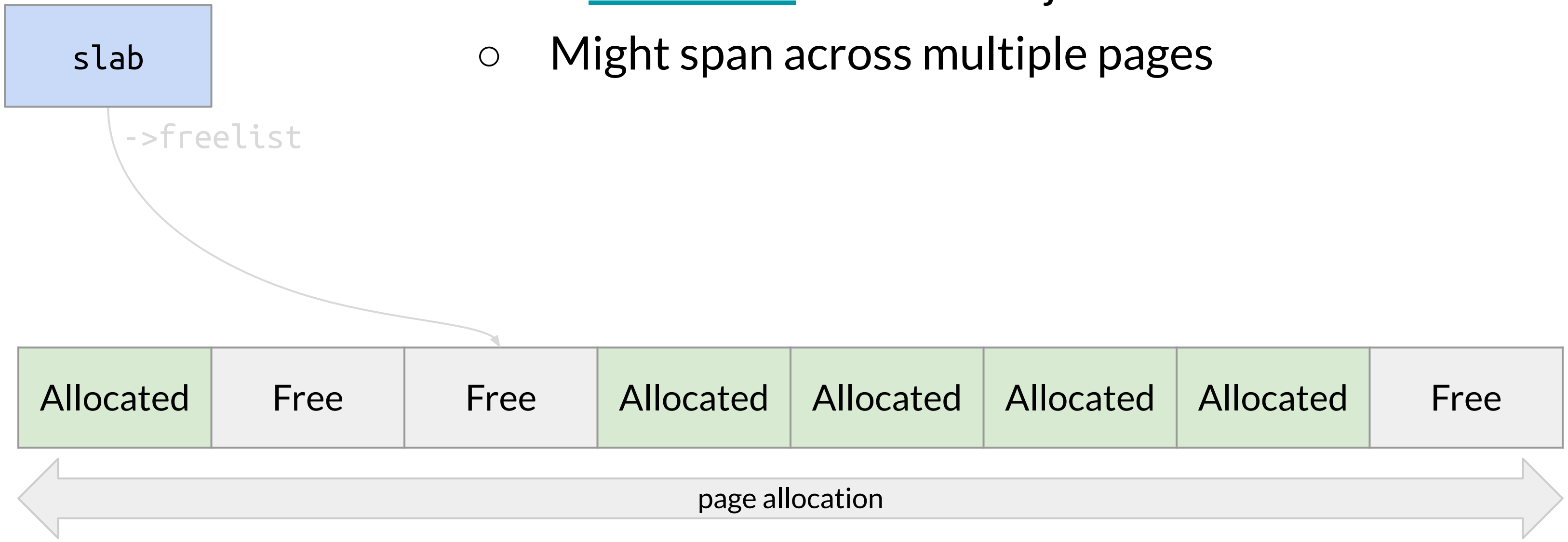
Each slab has `slab` structure

```
struct slab {  // Aliased with struct page
	struct kmem_cache *slab_cache; // Cache this slab belongs to
	struct slab *next;             // Next slab in per-cpu list
	int slabs;                     // Slabs left in per-cpu list
	struct list_head slab_list;    // List links in per-node list
	void *freelist;                // Per-slab freelist
	...
};
```



Each slab has backing memory

- Backing memory allocated via page_alloc
- Contains object slots
- Size is calculated based on object size
    - Might span across multiple pages



`/proc/slabinfo` shows sizes of slabs for caches

# name	<active_objs>	<num_objs>	<objsize>	<objperslab>	<pagesperslab>
cred_jar	7644	7644	192	21	1
kmalloc-8k	456	460	8192	4	8
kmalloc-4k	3118	3160	4096	8	8
kmalloc-2k	3621	3696	2048	16	8
kmalloc-32	54789	55808	32	128	1

- `pagesperslab` — size of slab backing memory for particular cache in pages
- `objsize` — size of objects, `objperslab` — number of objects in each slab

`struct slab` has `freelist` pointer

```
struct slab {  // Aliased with struct page
    void *freelist;  // Per-slab freelist
    ...
};
```

slab

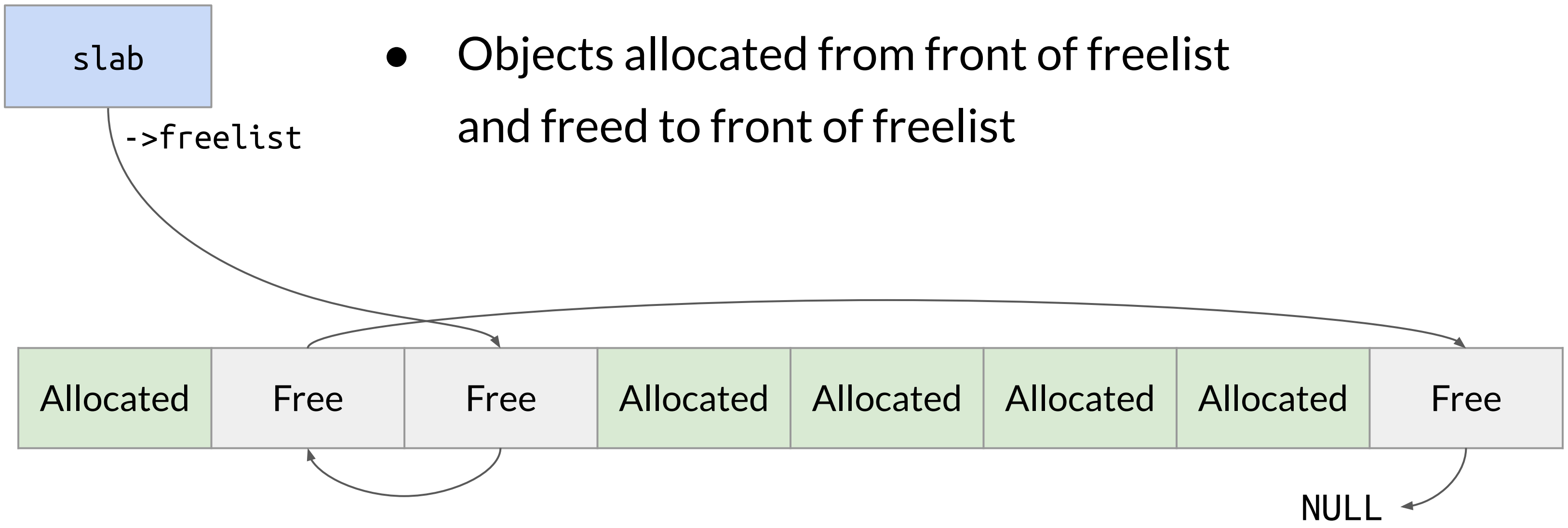
->freelist

- `freelist` points to first free slot within slab
    - There is no pointer to start of backing memory
      (no need for it, `struct slab` is `struct page`)

Allocated	Free	Free	Allocated	Allocated	Allocated	Allocated	Free

Free object slots are connected via freelist

- Pointer to first free slot is in `slab->freelist`
- Pointer to each next free slot is stored inside of free slot
- Pointer in last slot on freelist is `NULL`
- Objects allocated from front of freelist and freed to front of freelist



Freelist pointer in object slot

- Freelist pointer is stored [near middle](#) of object slot
  - For non-`SLAB_TYPESAFE_BY_RCU` caches without constructors
  - To prevent small-sized overflows corrupting freelist pointer

```
cache->offset = ALIGN_DOWN(cache->object_size / 2, sizeof(void *));
freeptr_addr = (unsigned long)object + cache->offset;
```

Small-sized overflow cannot reach freelist pointer

Freelist pointer is hashed

- Freelist pointer is hashed with `CONFIG_SLAB_FREELIST_HARDENED=y`
    - To make it hard to fake freelist pointer by overwriting it

```
cache->random = get_random_long();
freelist_ptr = (void *)(
    (unsigned long)ptr ^ cache->random ^ swab(ptr_addr));

// ptr - actual value of freelist pointer
// ptr_addr - location where freelist pointer is stored
// swab() - exchanges adjacent even and odd bytes
```

Freelist pointer is not great target for OOB/UAF

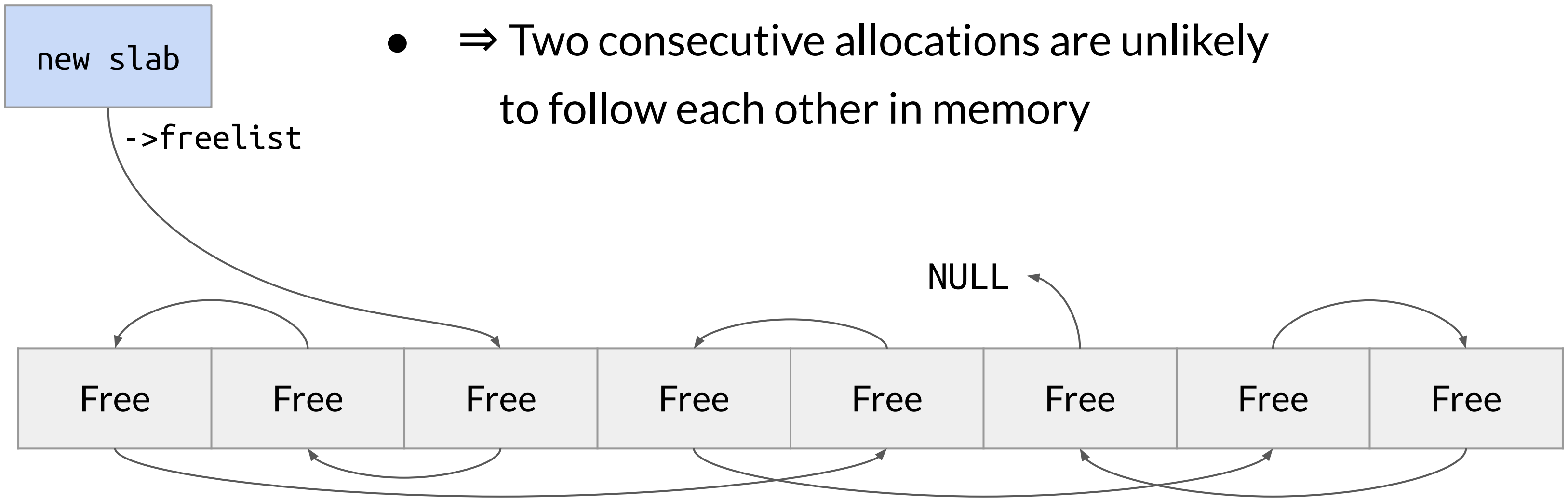
- Freelist pointer is hashed with `CONFIG_SLAB_FREELIST_HARDENED=y`
  - To make it hard to fake freelist pointer by overwriting it

- ⇒ Targeting freelist pointer via OOB/UAF is hard
  - Used to be common before `CONFIG_SLAB_FREELIST_HARDENED`
  - Technically still possible, but need to leak `cache->random` and `ptr_addr`

- ⇒ Most modern Slab exploits overwrite objects instead
  - Or other type of memory via cross-allocator attacks (out-of-scope)

Objects placed onto freelist in random order for new slabs

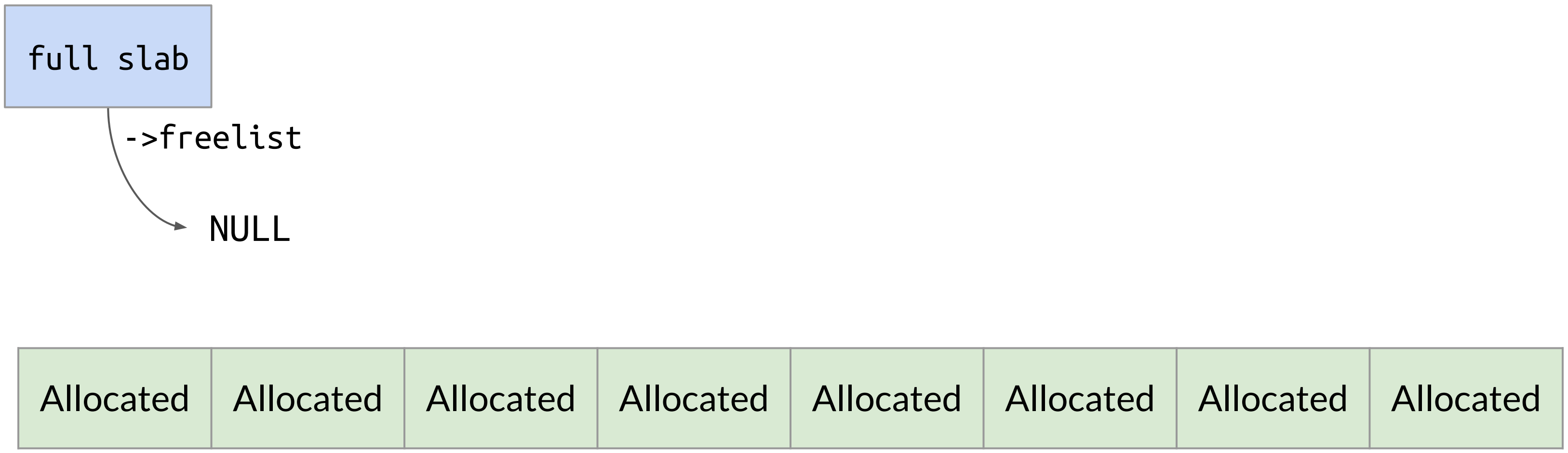
- SLUB shuffles order of objects on freelist when new slab is allocated
- With `CONFIG_SLAB_FREELIST_RANDOM=y`
- ⇒ Two consecutive allocations are unlikely to follow each other in memory



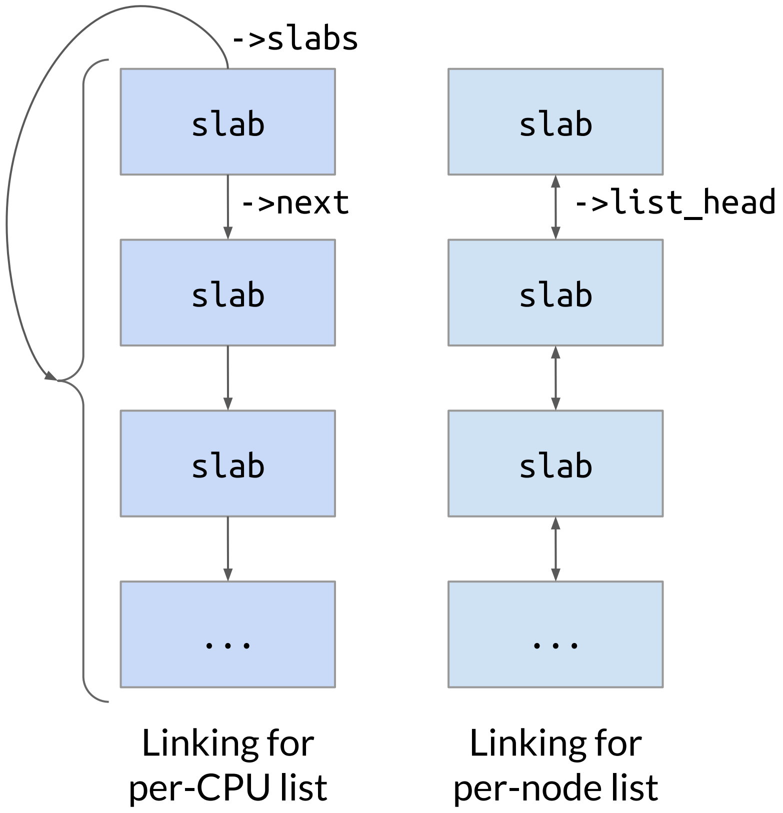
Freelist pointer is `NULL` for full slabs

- Once all objects in slab are allocated,

  `slab->freelist == NULL`



Multiple slabs can be linked together

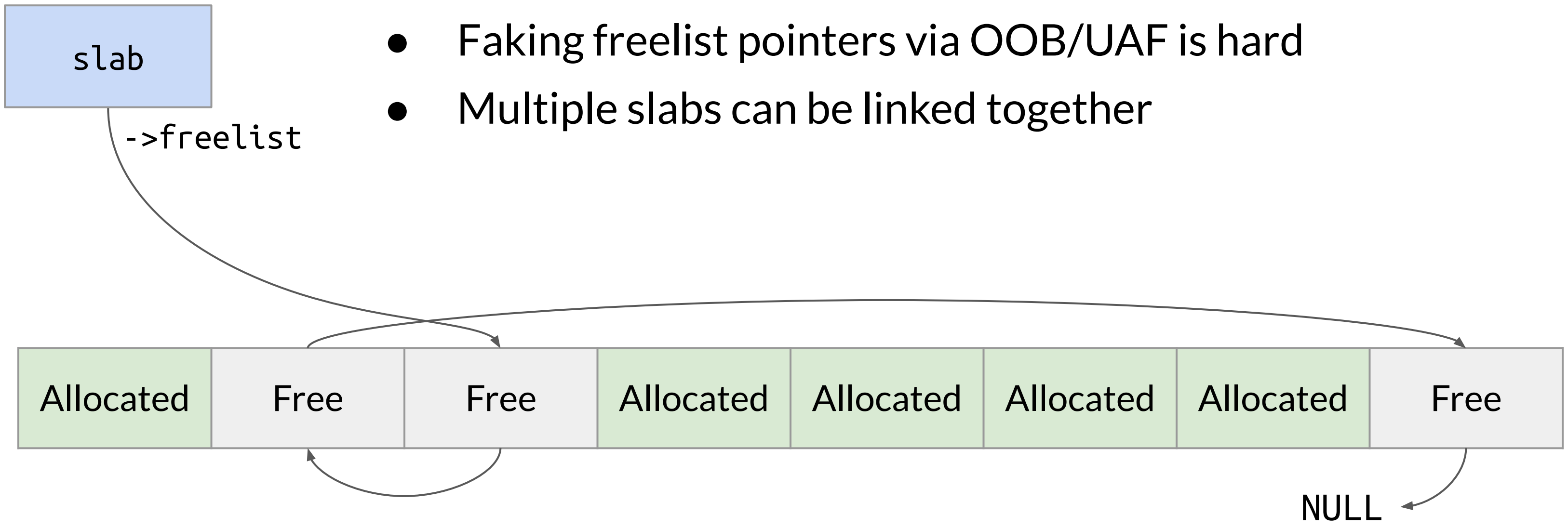



```
struct slab {
	struct slab *next;
	int slabs;
	struct list_head slab_list;
	...
};
```

- `next` and `slabs` used for per-CPU list
- `slab_list` used for per-node list

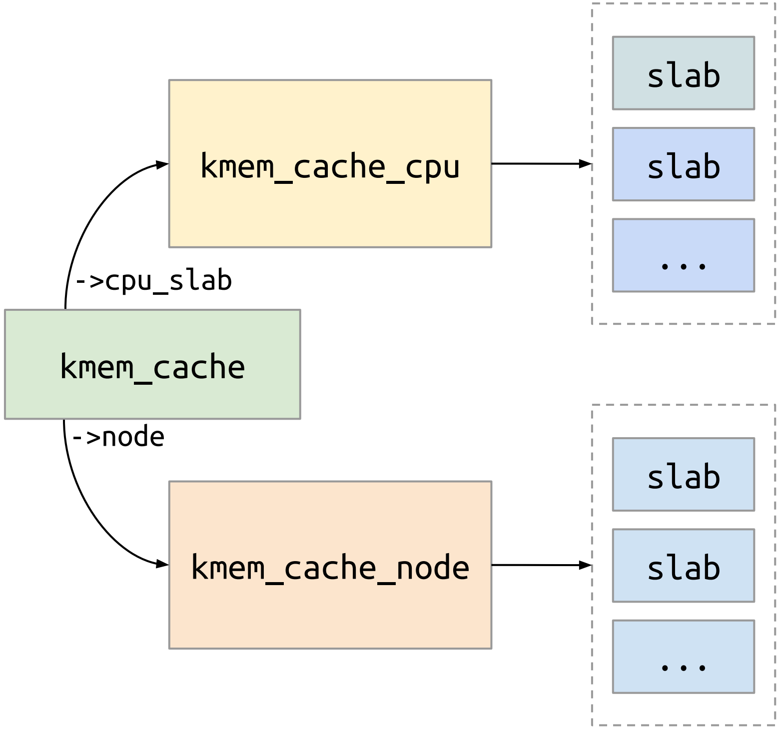
`struct slab` and its memory: Summary

- Each slab has `struct slab`, aliased with `struct page`
- Each slab has backing memory, allocated via page_alloc
- Free slots within slab linked via freelist in unknown order
- Faking freelist pointers via OOB/UAF is hard
- Multiple slabs can be linked together



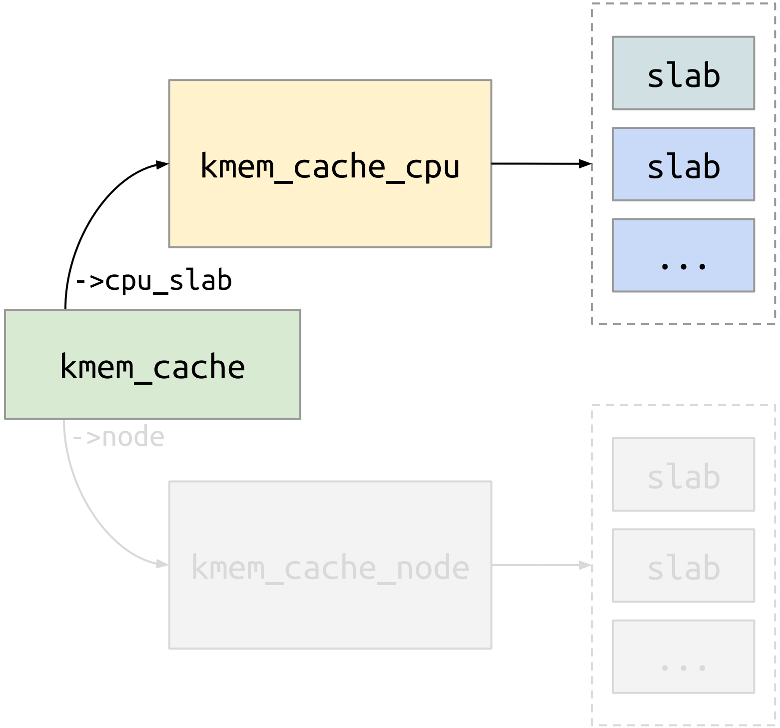
SLUB internals: Back to kmem_cache

kmem_cache_cpu/node contain pointers to slabs




```
struct kmem_cache_cpu {
	struct slab *slab;    // Active slab
	struct slab *partial; // Partial slabs
	...
};

struct kmem_cache_node {
	struct list_head partial; // Slabs
	...
};
```

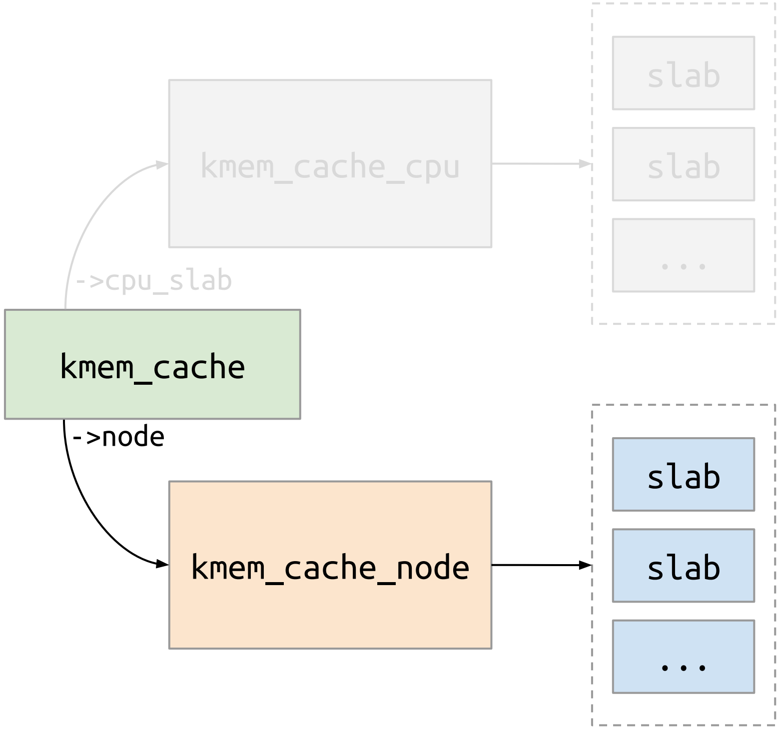
kmem_cache_cpu contains pointers to per-CPU slabs




```
struct kmem_cache_cpu {
	struct slab *slab;    // Active slab
	struct slab *partial; // Partial slabs
	...
};
```

- Per-CPU slabs bound to particular CPU
  - Aka frozen slabs (note change in 6.8)
- Allocations on that CPU will happen from these slabs first
- Done for performance reasons (locking, CPU caches, etc.)

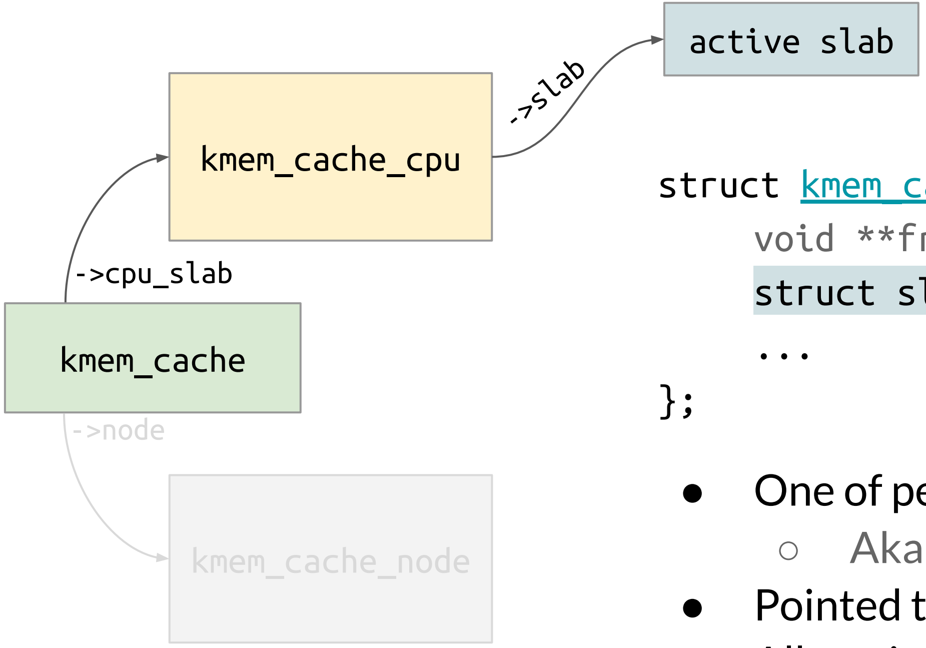
kmem_cache_node contains pointer to per-node slabs




```
struct kmem_cache_node {
	struct list_head partial; // Slabs
	...
};
```

- Per-node slabs have backing memory coming from corresponding NUMA node
- Not bound to any CPU yet
- But still belong to cache (contain objects allocated from it)

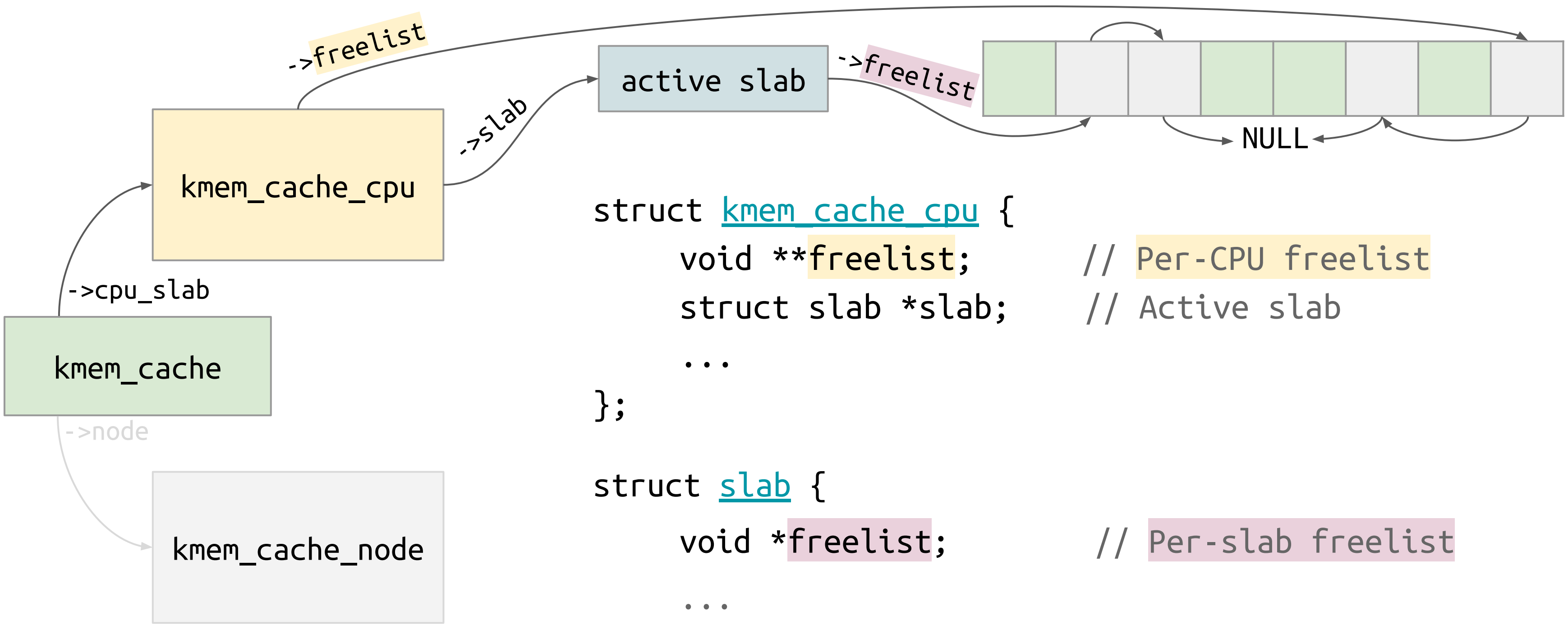
kmem_cache_cpu has one active slab




```
struct kmem_cache_cpu {
	void **freelist;	// Per-CPU freelist
	struct slab *slab;	// Active slab
	...
};
```

- One of per-CPU slabs is designated active
  - Aka "the CPU slab" in SLUB source code
- Pointed to by `kmem_cache_cpu->slab`
- Allocations served from this slab first

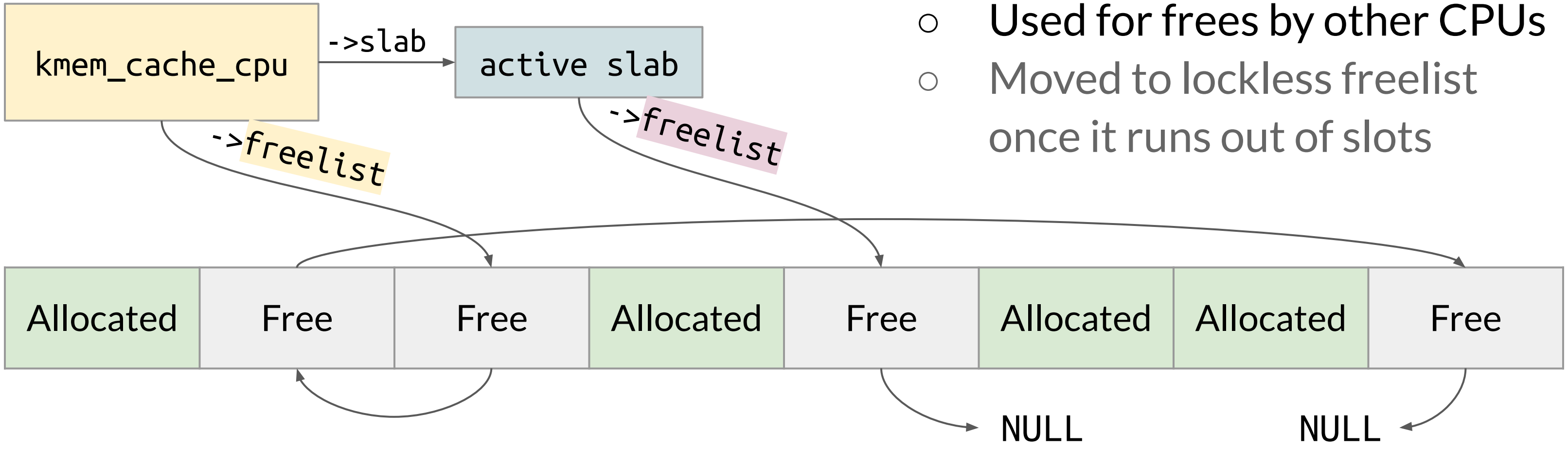
Active slab has two freelists



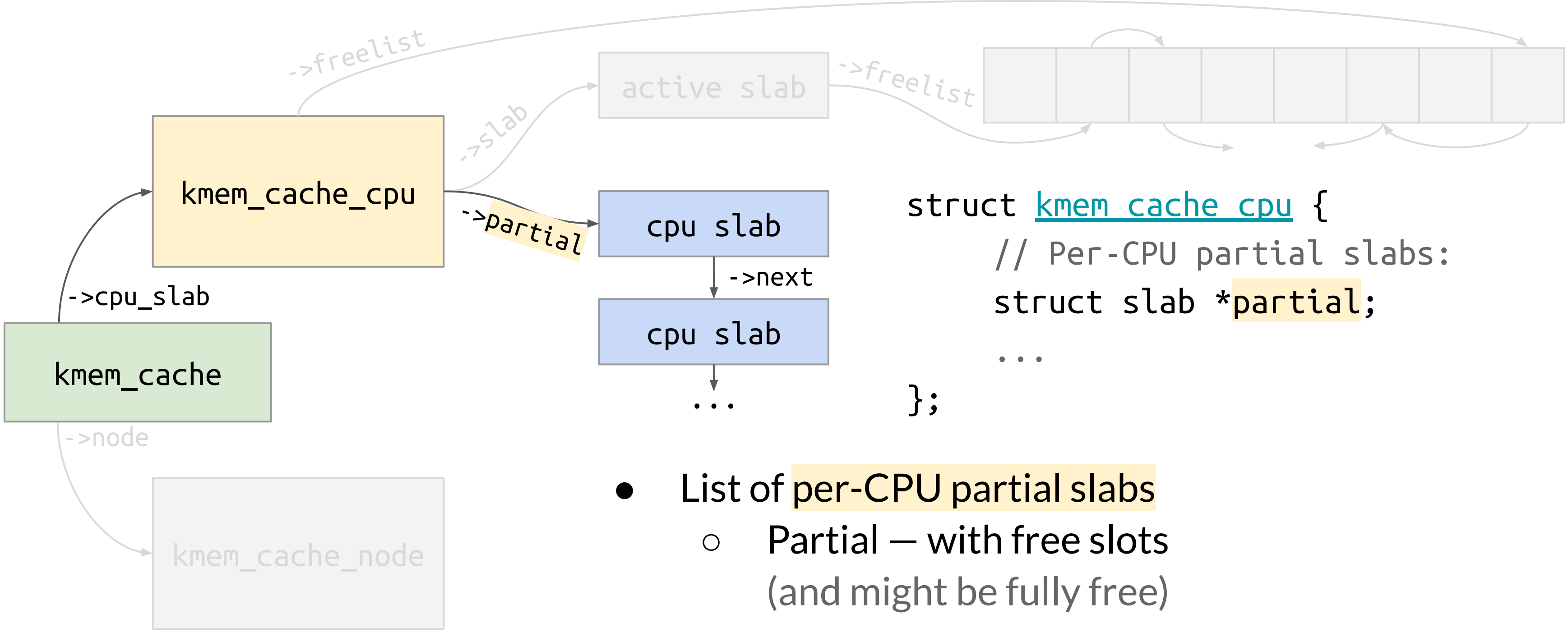

```
struct kmem_cache_cpu {
	void **freelist;        // Per-CPU freelist
	struct slab *slab;      // Active slab
	...
};

struct slab {
	void *freelist;         // Per-slab freelist
	...
};
```

Two freelists of active slab

- Lockless per-CPU freelist `kmem_cache_cpu->freelist`
  - Used for allocations and frees by this CPU
- Per-slab freelist for active slab `kmem_cache_cpu->slab->freelist`
  - Used for frees by other CPUs
  - Moved to lockless freelist once it runs out of slots



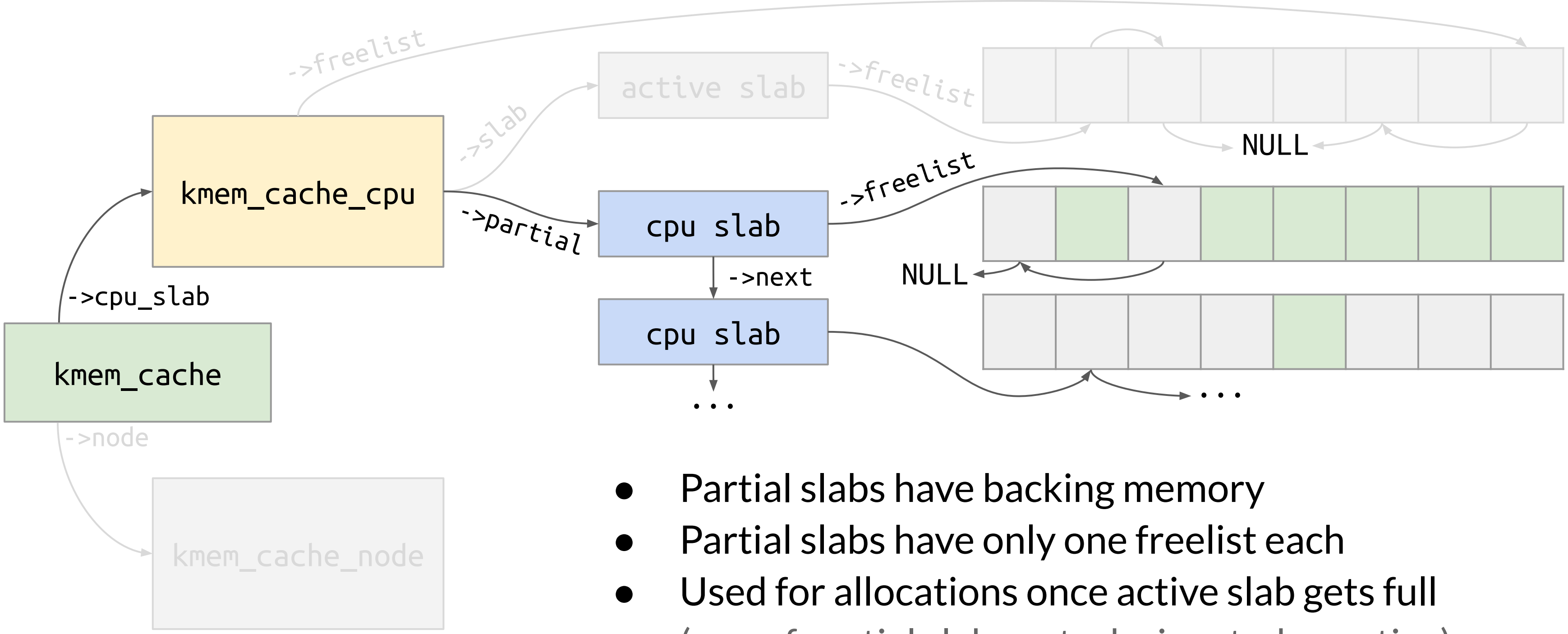
kmem_cache_cpu has list of per-CPU partial slabs [1/2]




```
struct kmem_cache_cpu {
	// Per-CPU partial slabs:
	struct slab *partial;
	...
};
```

- List of per-CPU partial slabs
  - Partial — with free slots
    (and might be fully free)

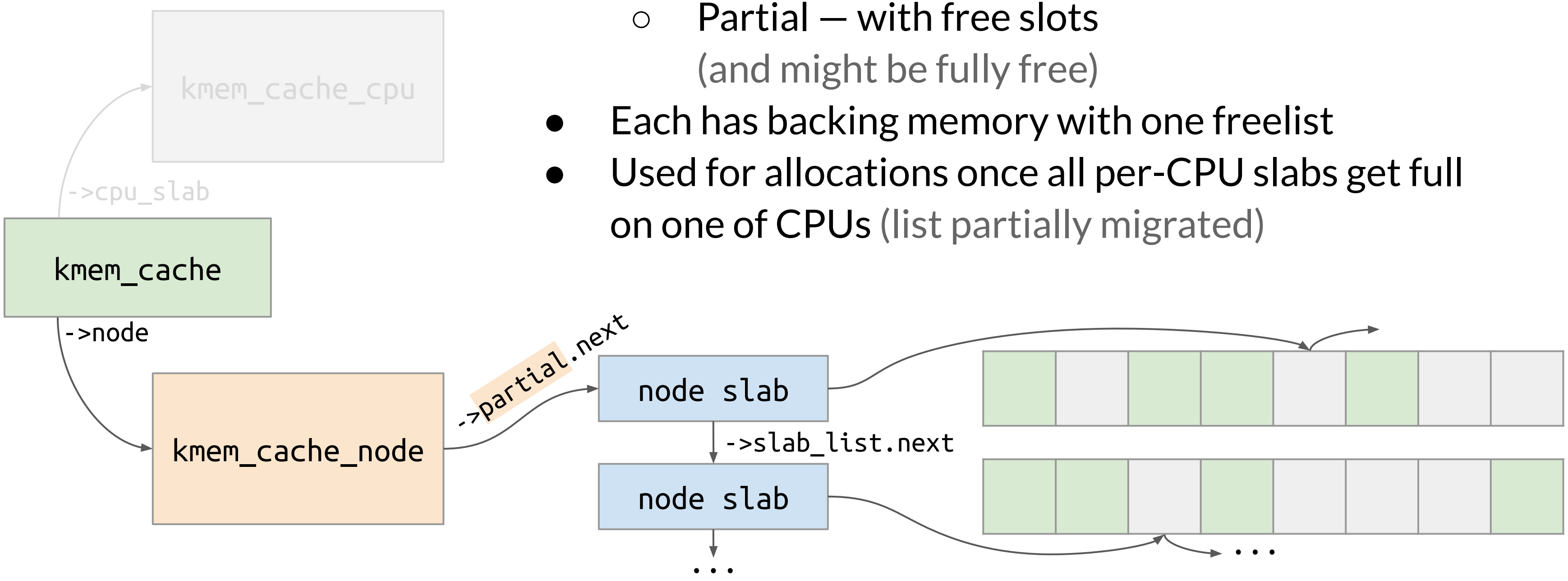
kmem_cache_cpu has list of per-CPU partial slabs [2/2]



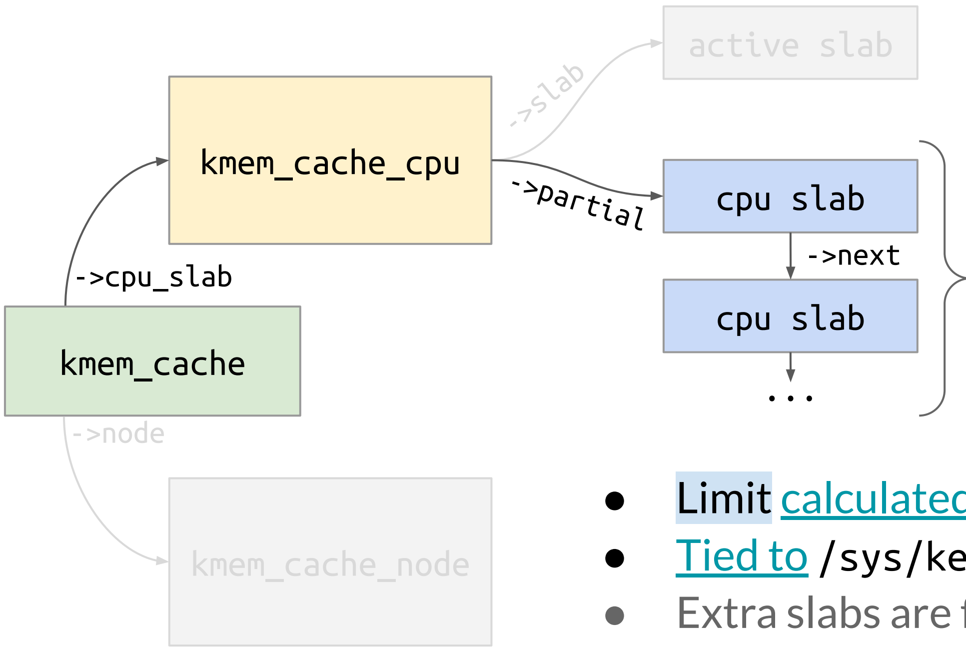
- Partial slabs have backing memory
- Partial slabs have only one freelist each
- Used for allocations once active slab gets full (one of partial slabs gets designated as active)

kmem_cache_node has list of per-node partial slabs

- List of per-node partial slabs
  - Partial — with free slots
    (and might be fully free)
- Each has backing memory with one freelist
- Used for allocations once all per-CPU slabs get full on one of CPUs (list partially migrated)



Maximum number of kept per-CPU partial slabs is limited

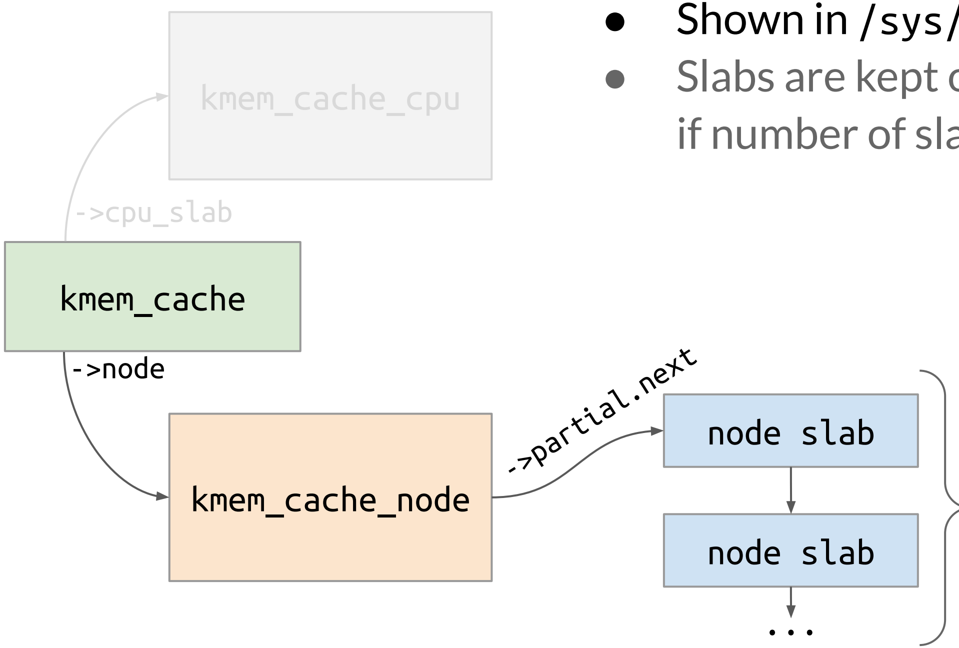



```
struct kmem_cache {
	unsigned int cpu_partial;
	unsigned int cpu_partial_slabs;
	...
};
```

- Limit calculated based on object and slab sizes
- Tied to /sys/kernel/slab/$CACHE/cpu_partial
- Extra slabs are freed (if possible)
  or moved to per-node list (discussed later)

Minimum number of kept per-node slabs is maintained

- Number calculated based on object size
- Shown in `/sys/kernel/slab/$CACHE/min_partial`
- Slabs are kept on list even if fully free if number of slabs is <= limit




```
struct kmem_cache {
      unsigned long min_partial;
      ...
};
```

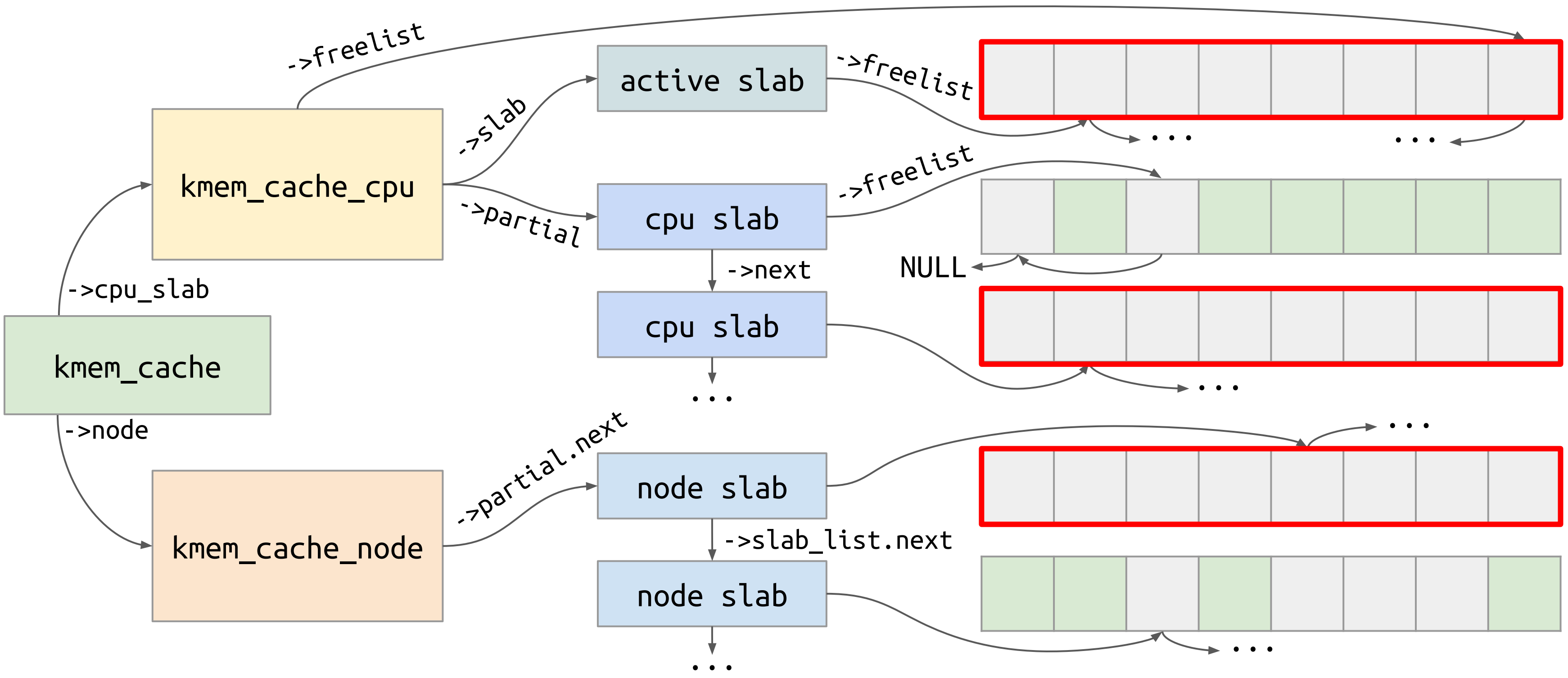
Try to keep at least this many

Limits for `kmalloc` caches

Cache	`cpu_partial`	`cpu_partial_slabs`	`min_partial`
`kmalloc-8`, `kmalloc-16`	120	1	5
`kmalloc-32`	120	2	5
`kmalloc-64`	120	4	5
`kmalloc-128`	120	8	5
`kmalloc-192`	120	12	5
`kmalloc-256`, `kmalloc-512`	52	7	5
`kmalloc-1k`, `kmalloc-2k`	24	3	5
`kmalloc-4k`	6	2	6
`kmalloc-8k`	6	3	6

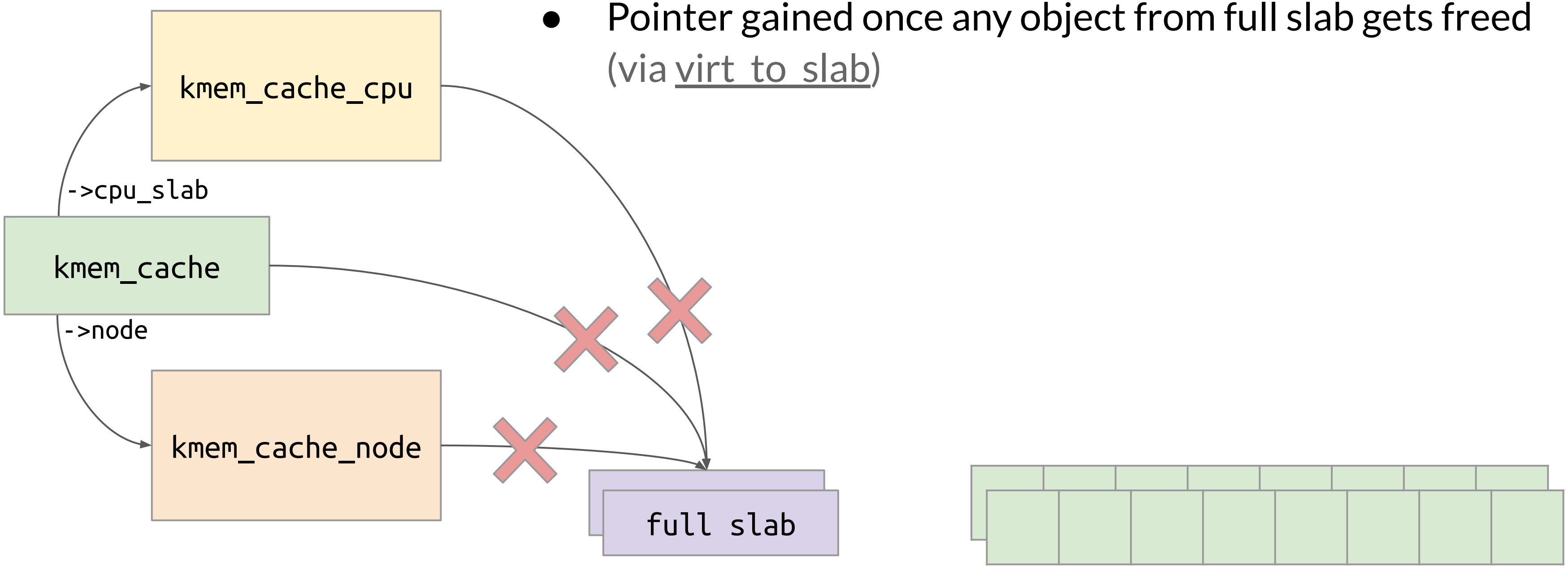
Per-CPU numbers changed in 5.16

Slab of each type might be fully empty

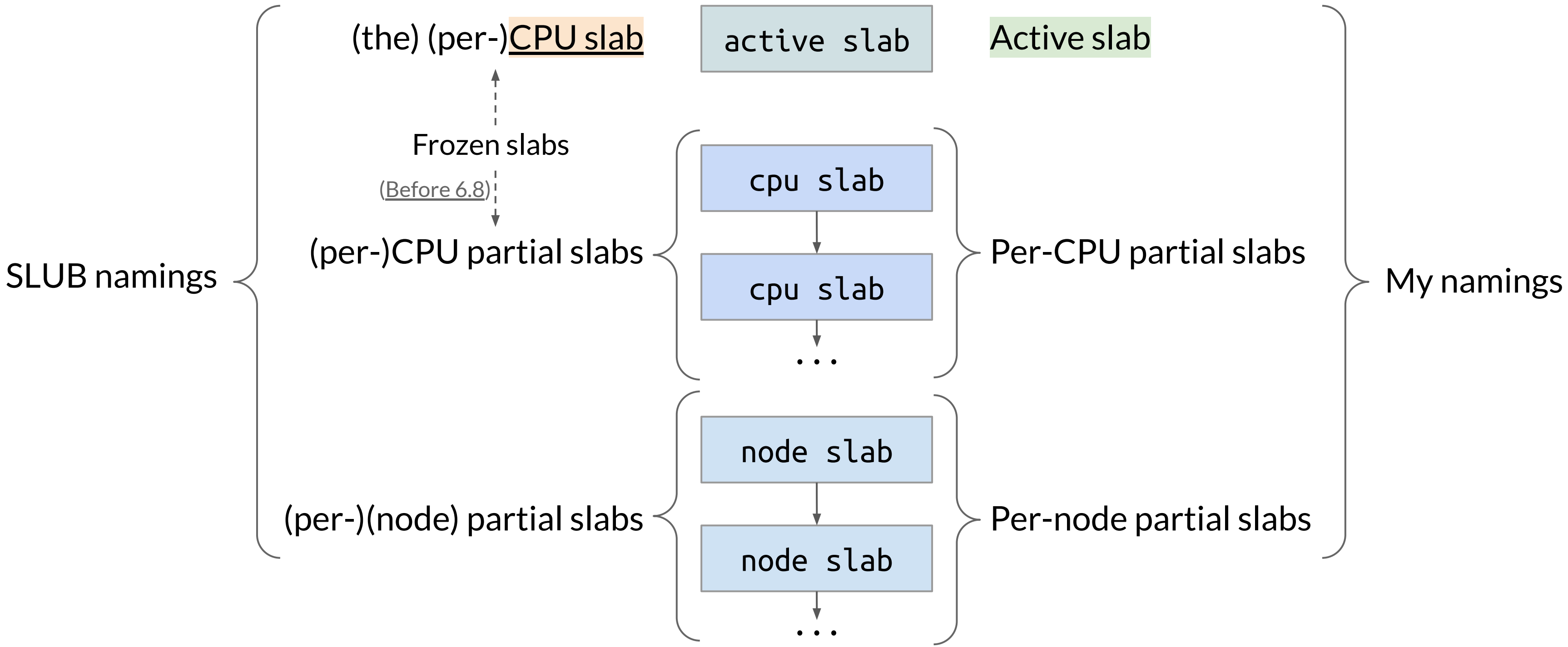


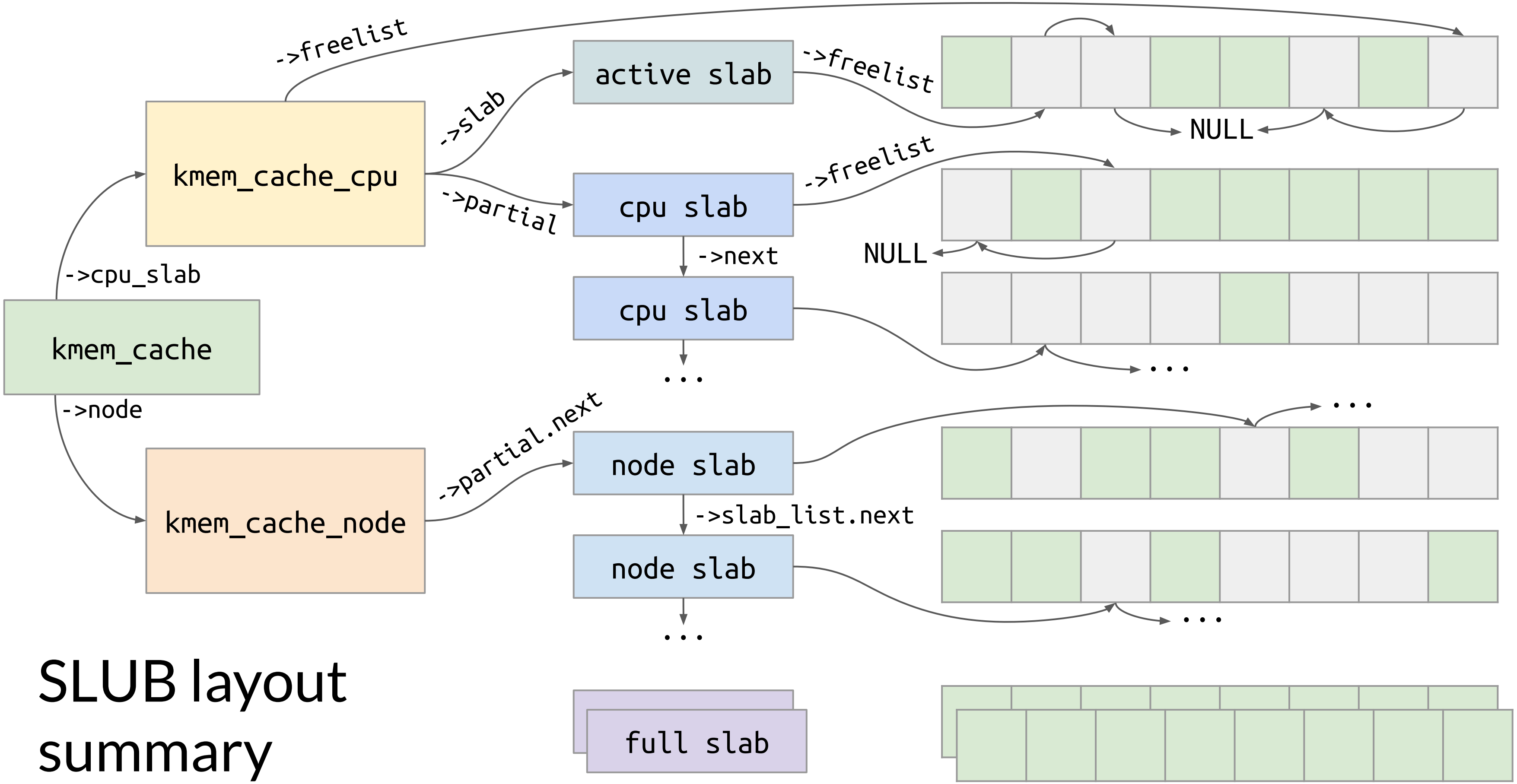
Full slabs are not tracked

- No pointers to full slabs (unless slub_debug is enabled)
- Pointer gained once any object from full slab gets freed (via virt_to_slab)



My vs SLUB namings for slab types

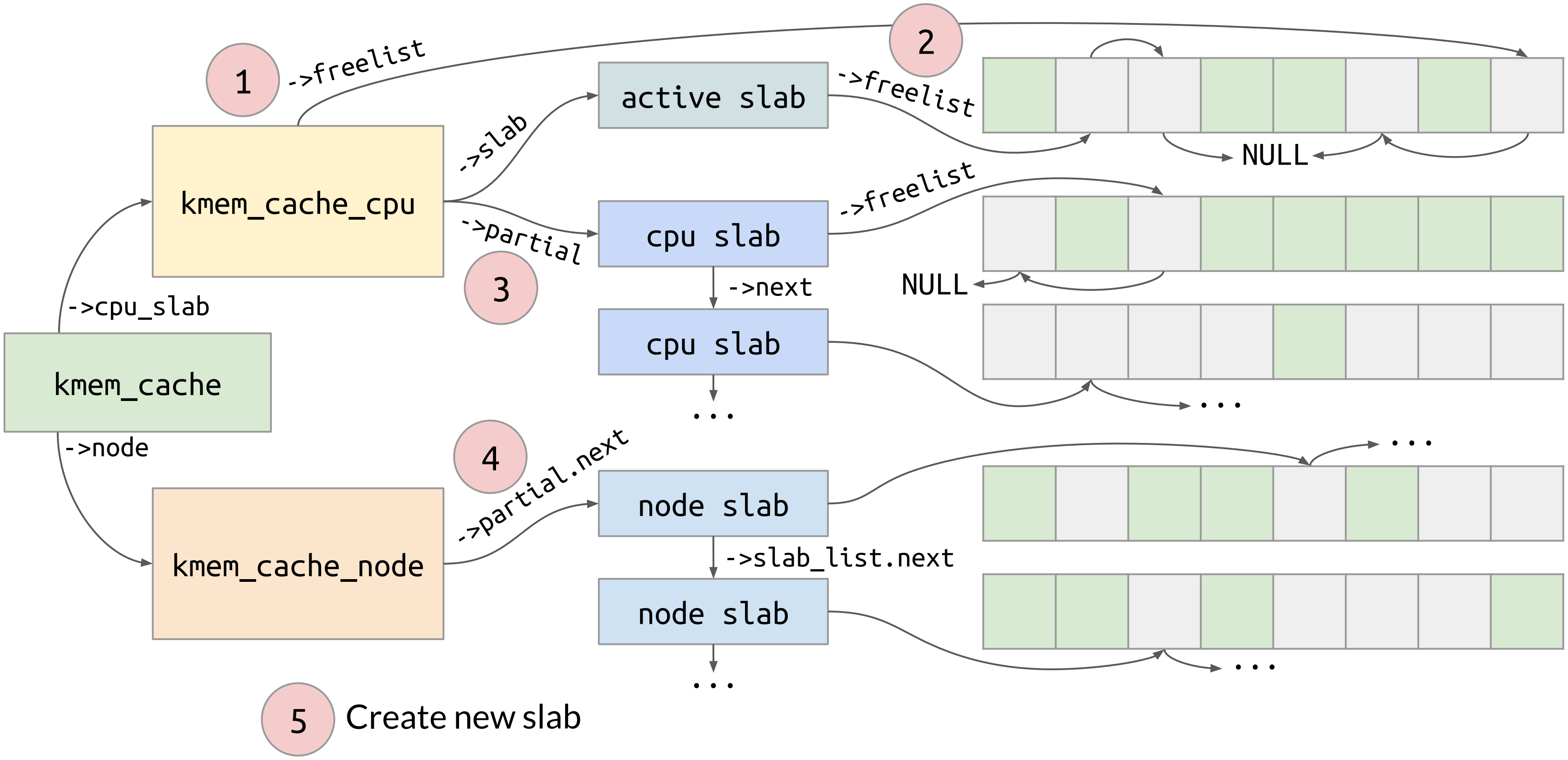




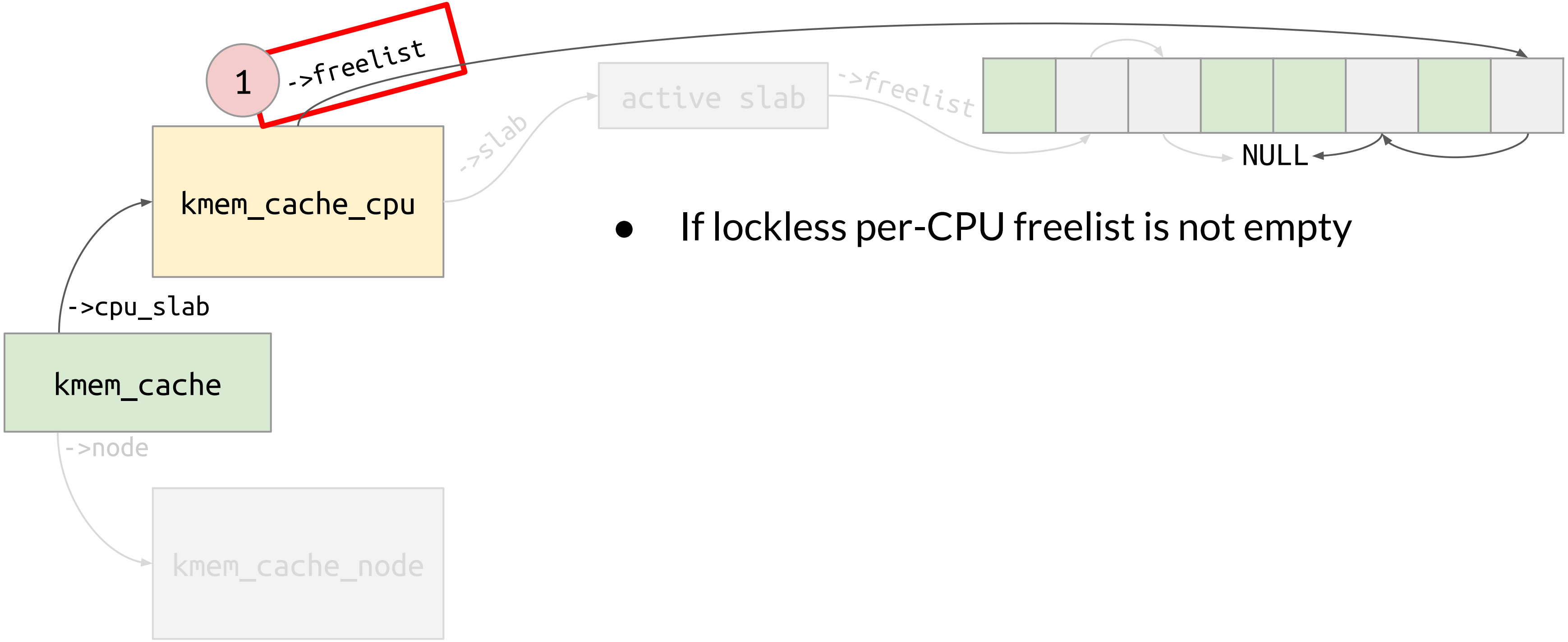
SLUB layout summary

SLUB internals:
Allocation process

Preview: 5 tiers of allocation process

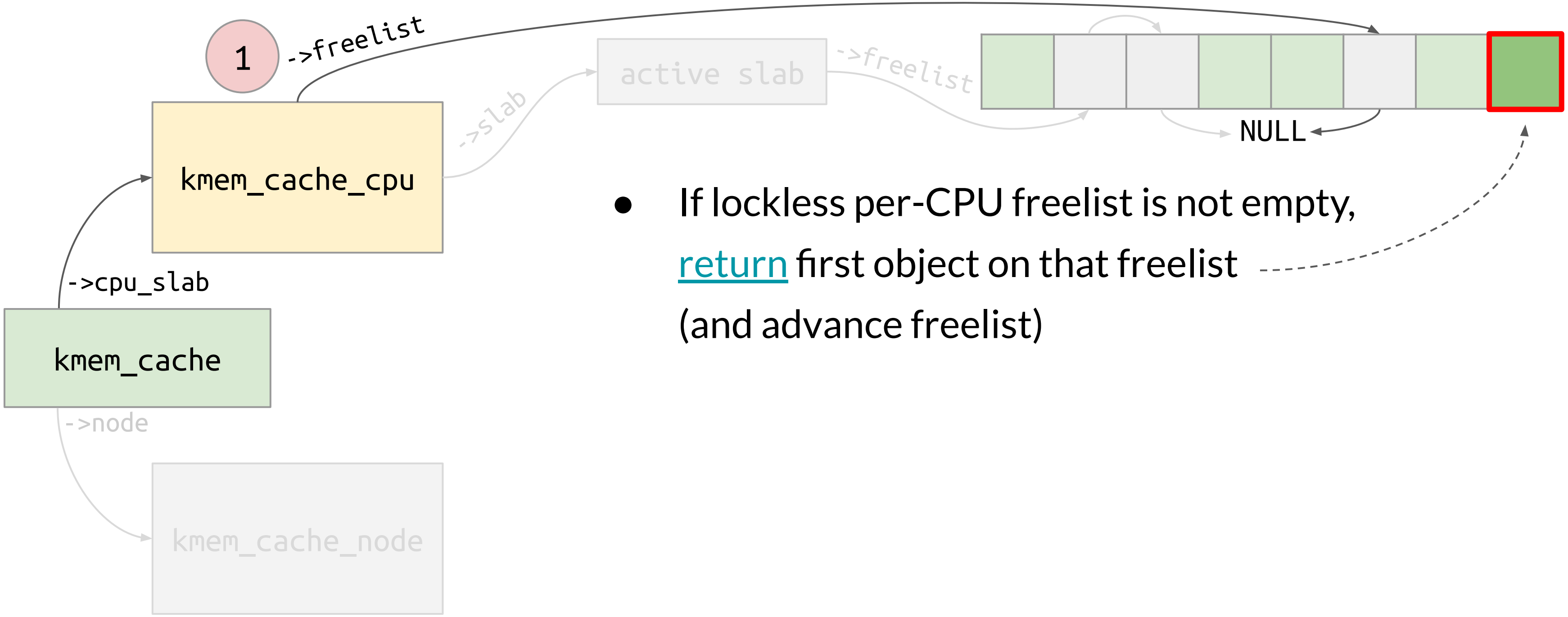


Tier 1: Allocating from lockless per-CPU freelist [0/2]



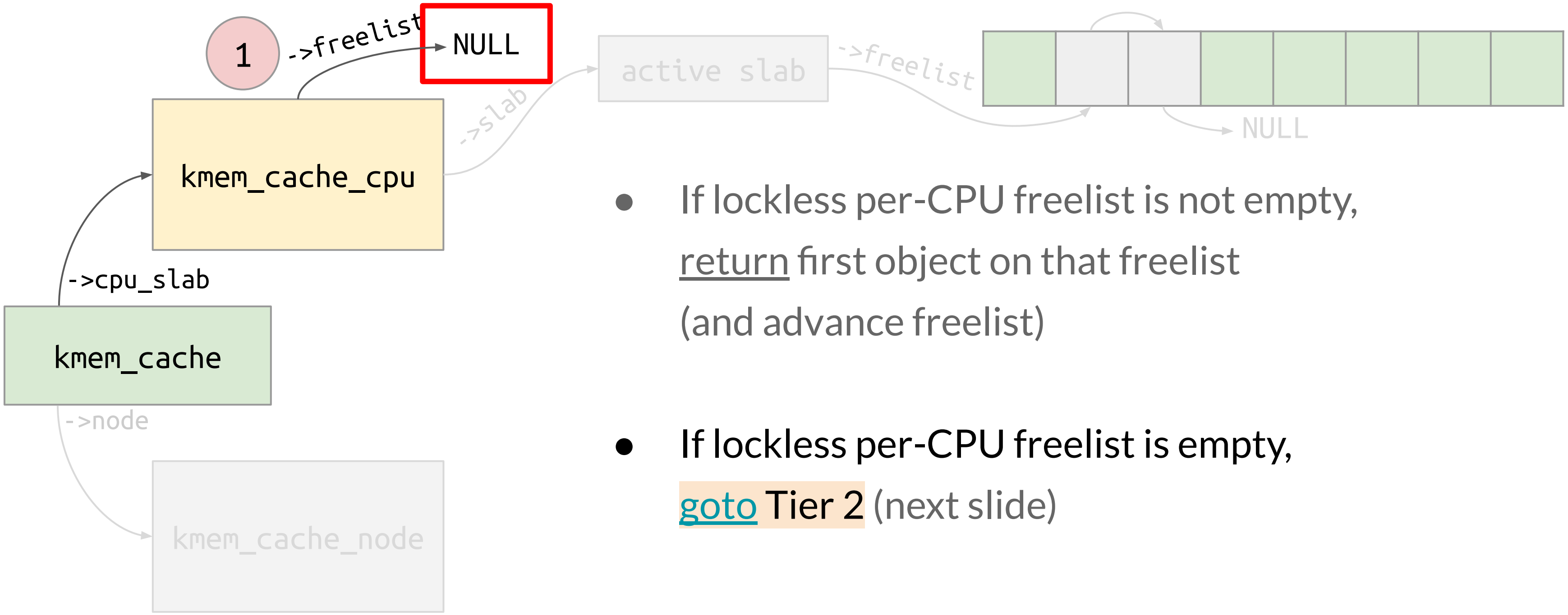
- If lockless per-CPU freelist is not empty

Tier 1: Allocating from lockless per-CPU freelist [1/2]



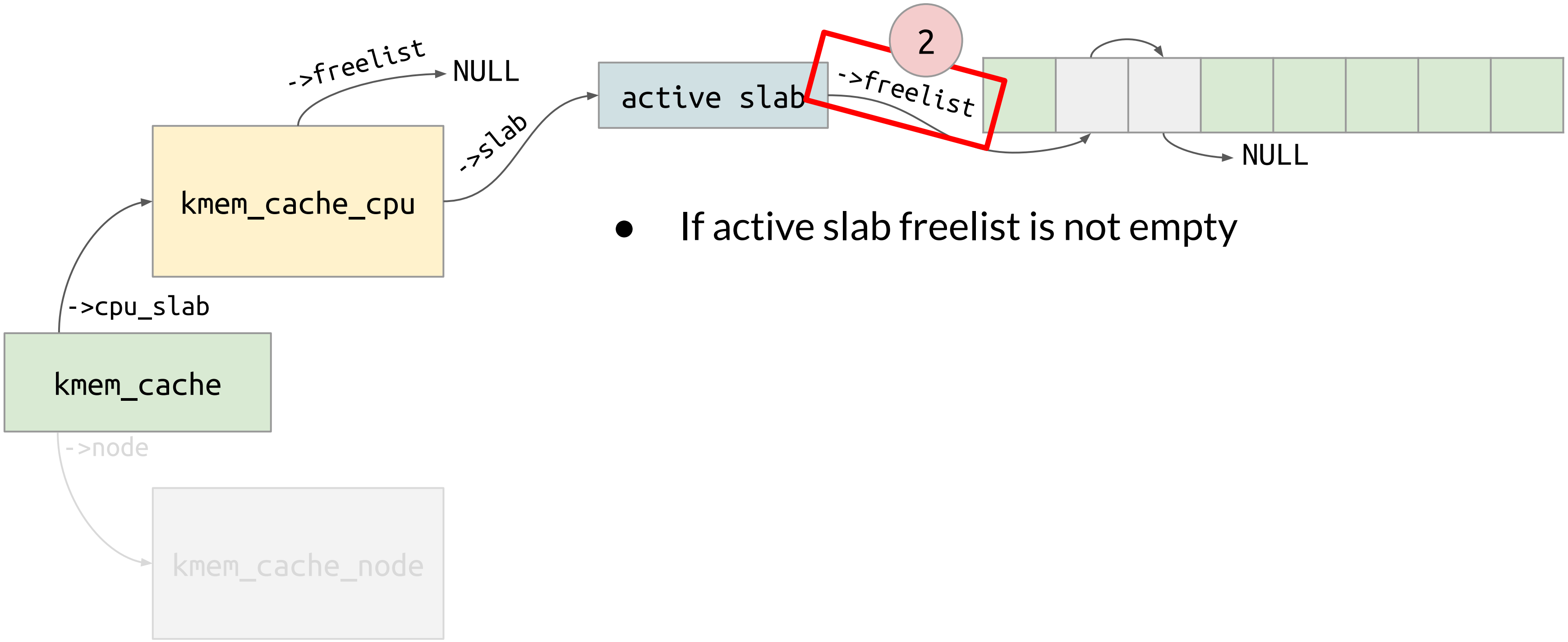
- If lockless per-CPU freelist is not empty, [return](#) first object on that freelist (and advance freelist)

Tier 1: Allocating from lockless per-CPU freelist [2/2]



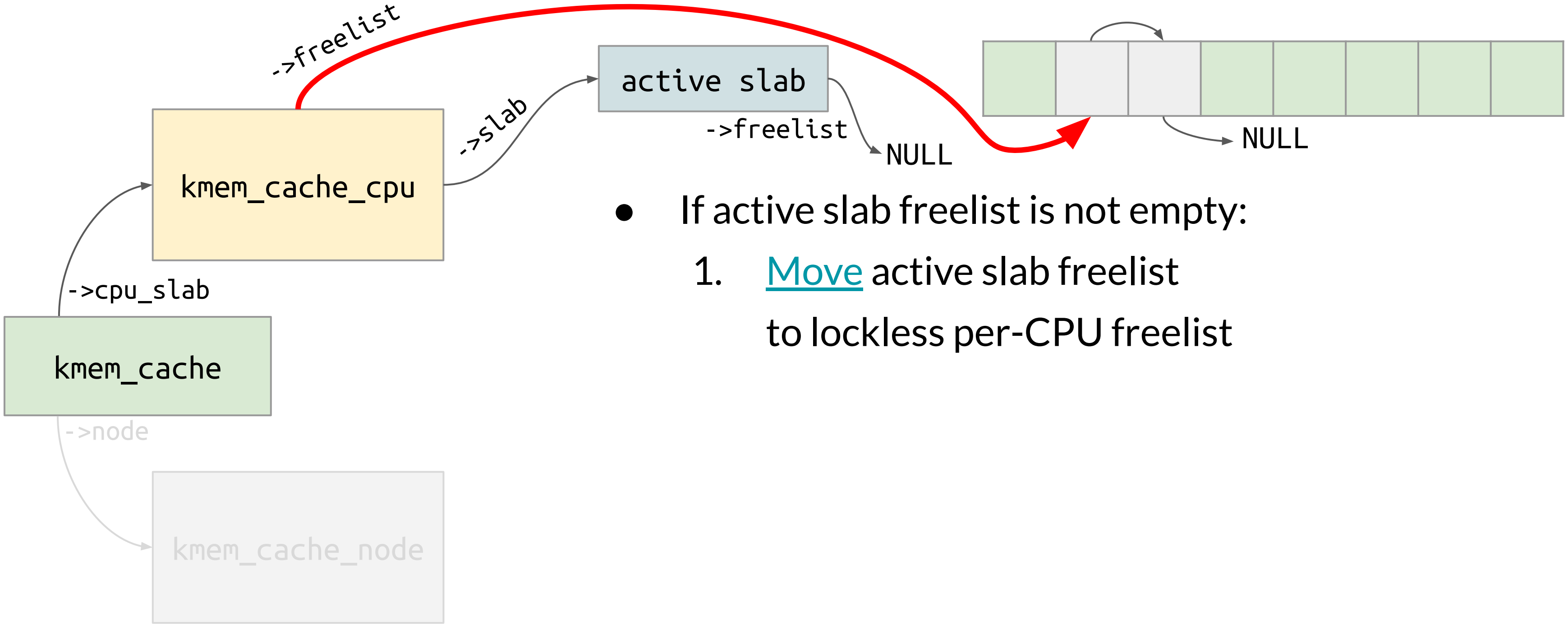
- If lockless per-CPU freelist is not empty, return first object on that freelist (and advance freelist)
- If lockless per-CPU freelist is empty, goto Tier 2 (next slide)

Tier 2: Allocating from active slab freelist [0/3]



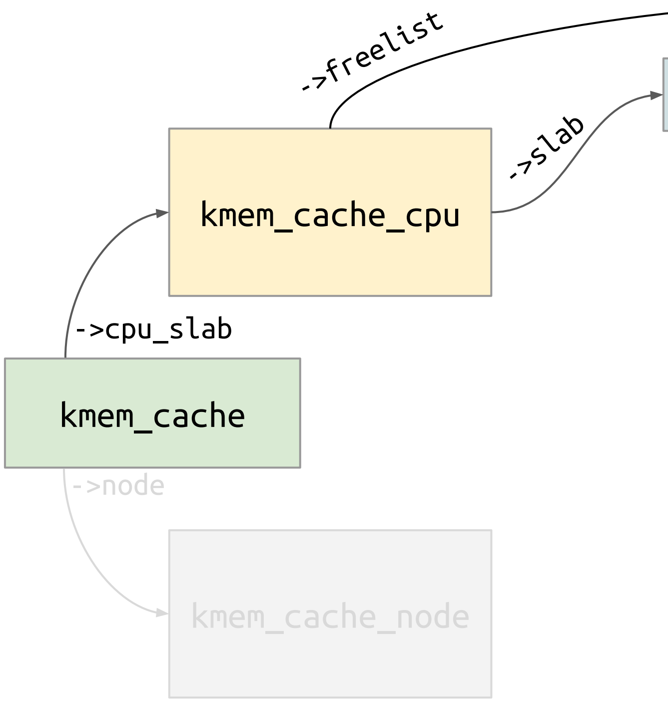
- If active slab freelist is not empty

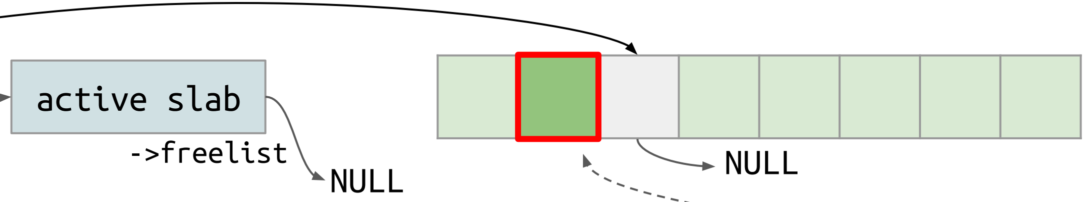
Tier 2: Allocating from active slab freelist [1/3]



- If active slab freelist is not empty:
    1. Move active slab freelist to lockless per-CPU freelist

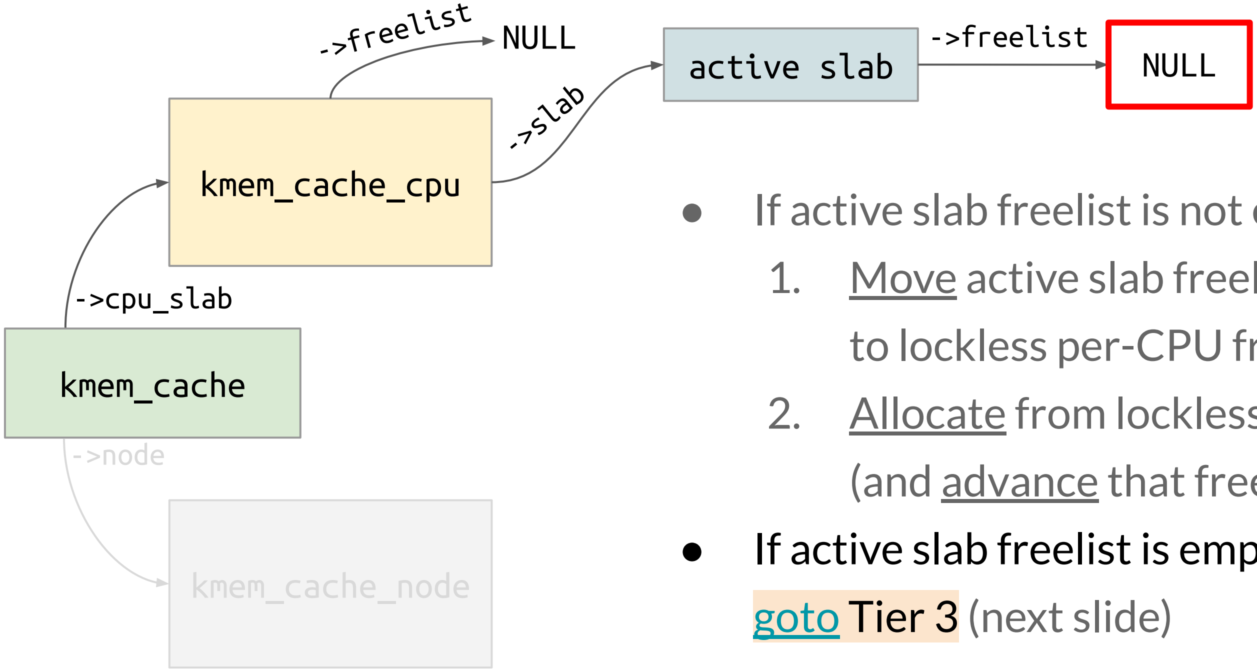
Tier 2: Allocating from active slab freelist [2/3]





- If active slab freelist is not empty:
  1. Move active slab freelist to lockless per-CPU freelist
  2. Allocate from lockless per-CPU freelist (and advance that freelist)

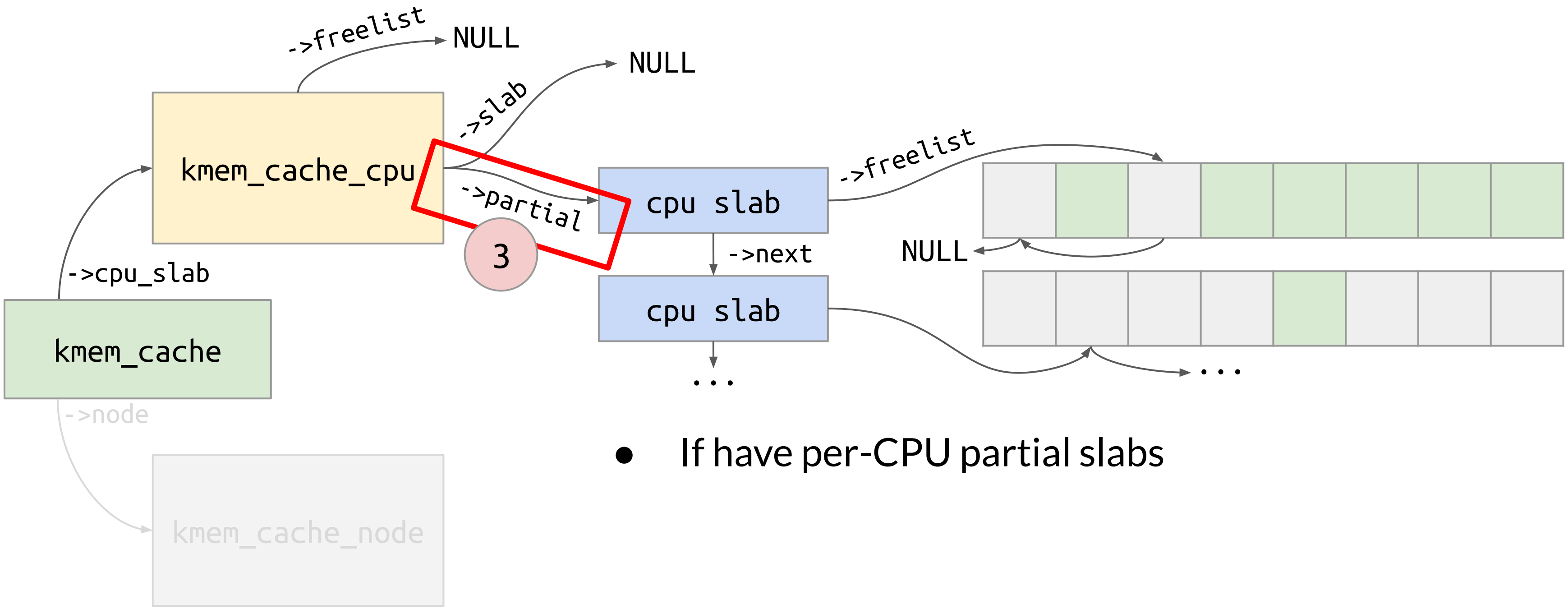
Tier 2: Allocating from active slab freelist [3/3]



Or `kmem_cache_cpu->slab == NULL`

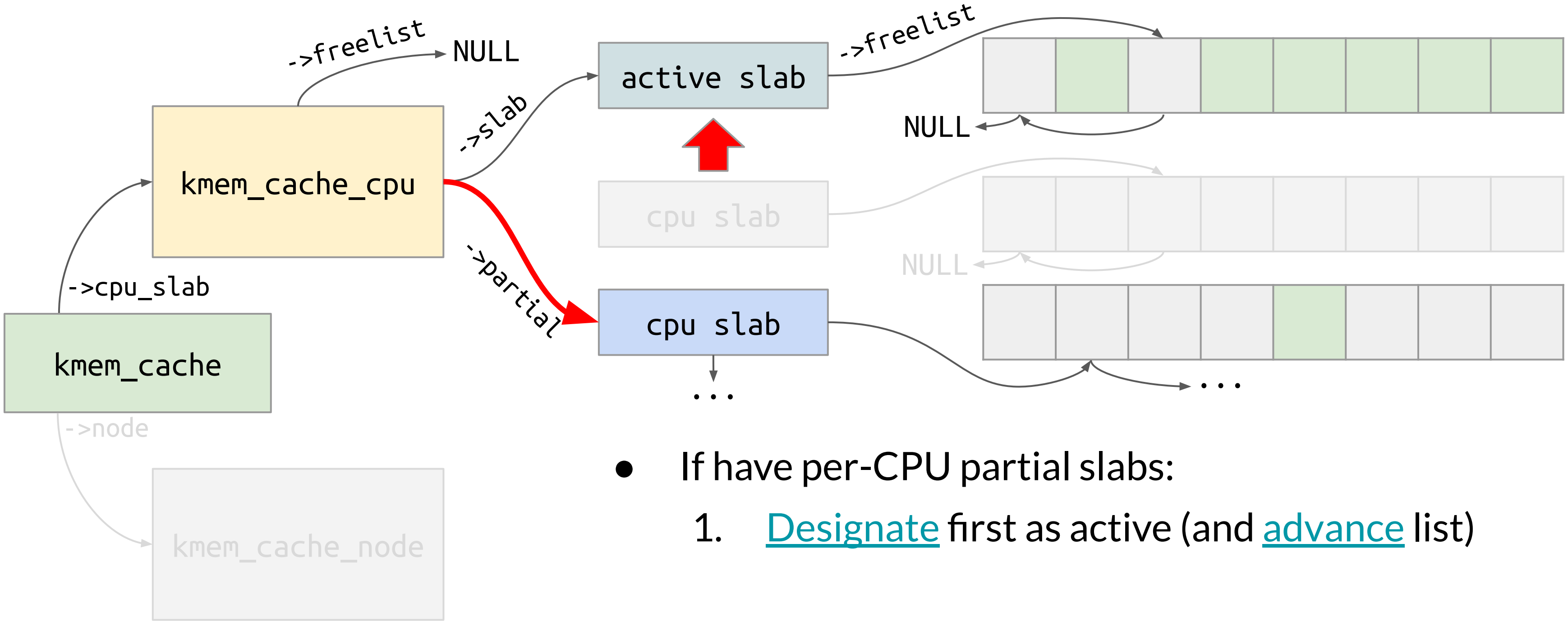
- If active slab freelist is not empty:
  1. Move active slab freelist to lockless per-CPU freelist
  2. Allocate from lockless per-CPU freelist (and advance that freelist)
- If active slab freelist is empty, goto Tier 3 (next slide)

Tier 3: Allocating from per-CPU partial slabs [0/3]



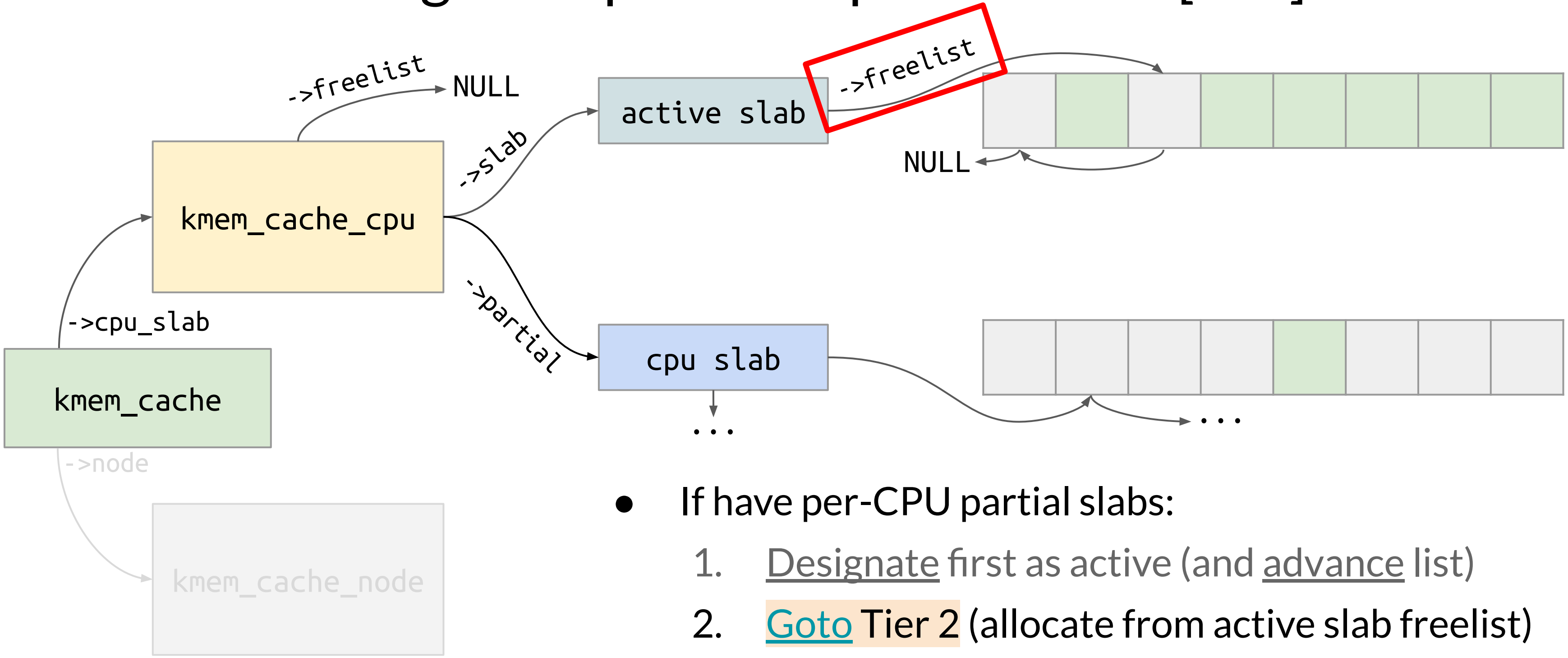
- If have per-CPU partial slabs

Tier 3: Allocating from per-CPU partial slabs [1/3]



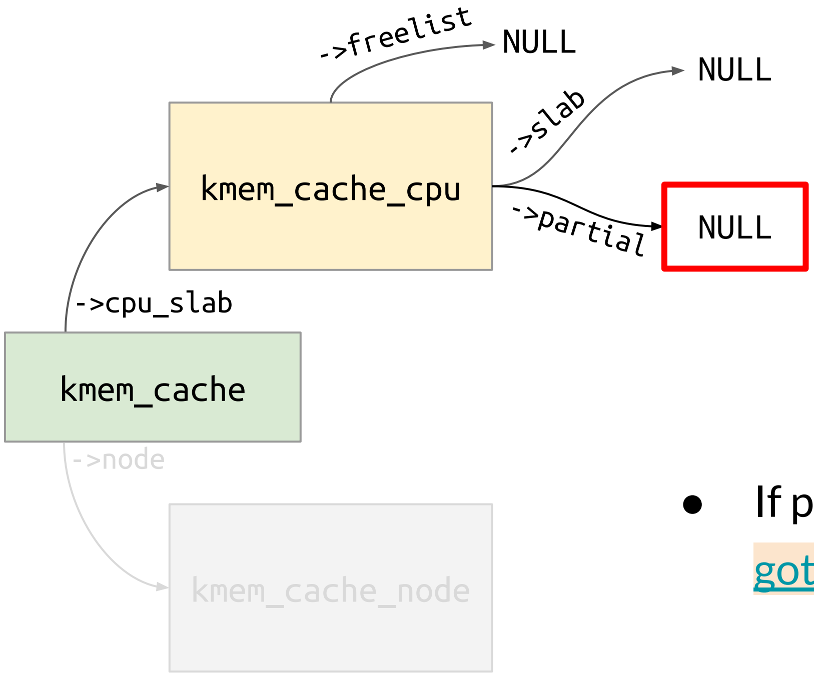
- If have per-CPU partial slabs:
  1. Designate first as active (and advance list)

Tier 3: Allocating from per-CPU partial slabs [2/3]



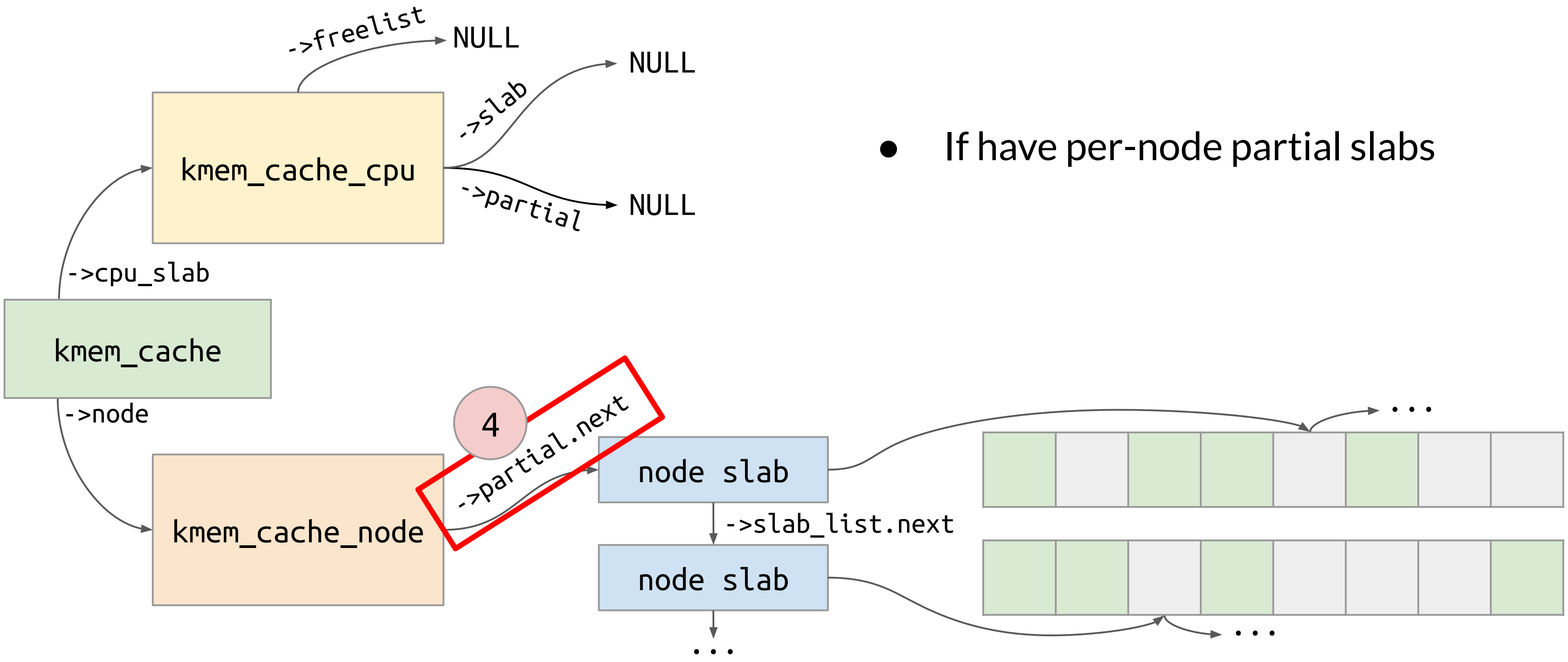
- If have per-CPU partial slabs:
    1. Designate first as active (and advance list)
    2. Goto Tier 2 (allocate from active slab freelist)

Tier 3: Allocating from per-CPU partial slabs [3/3]



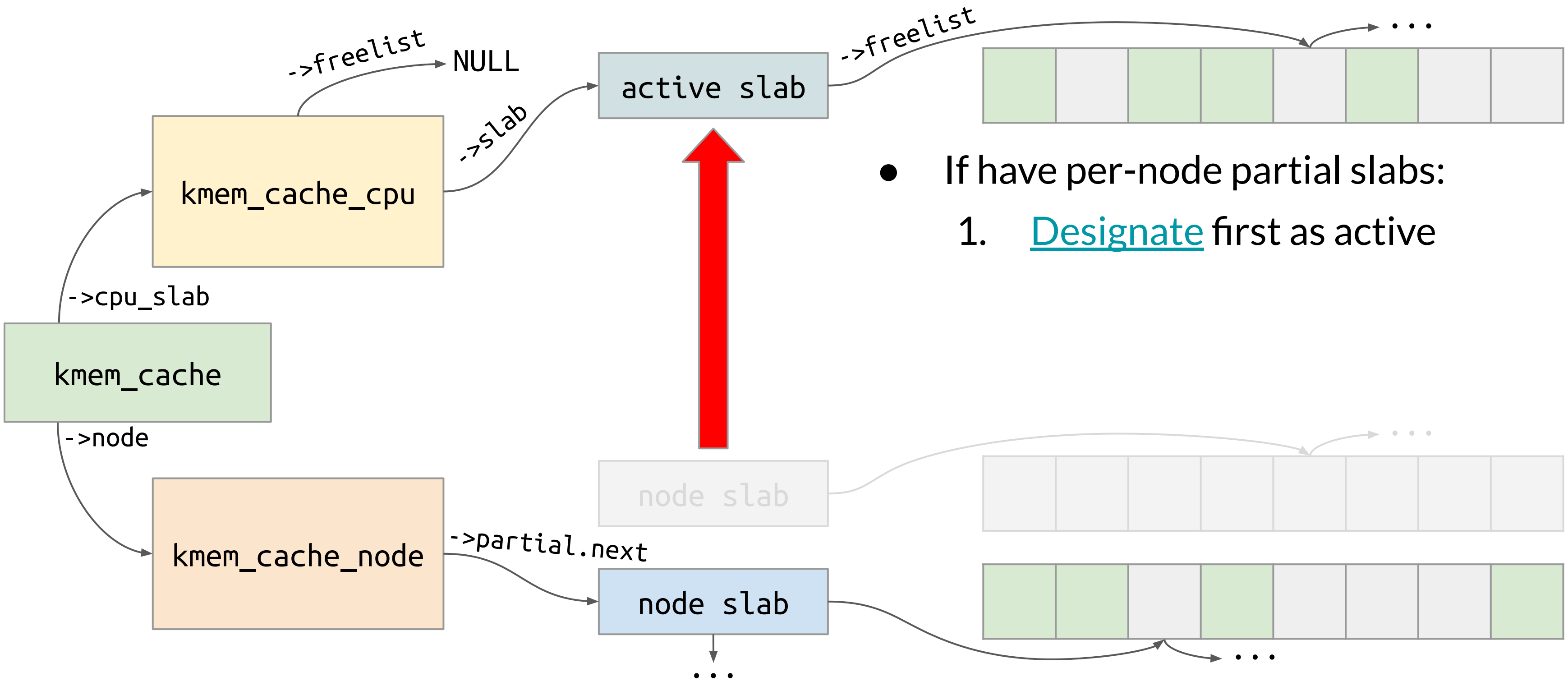
- If per-CPU partial slab list is empty, goto Tier 4 (next slide)

Tier 4: Allocating from per-node partial slabs [0/4]



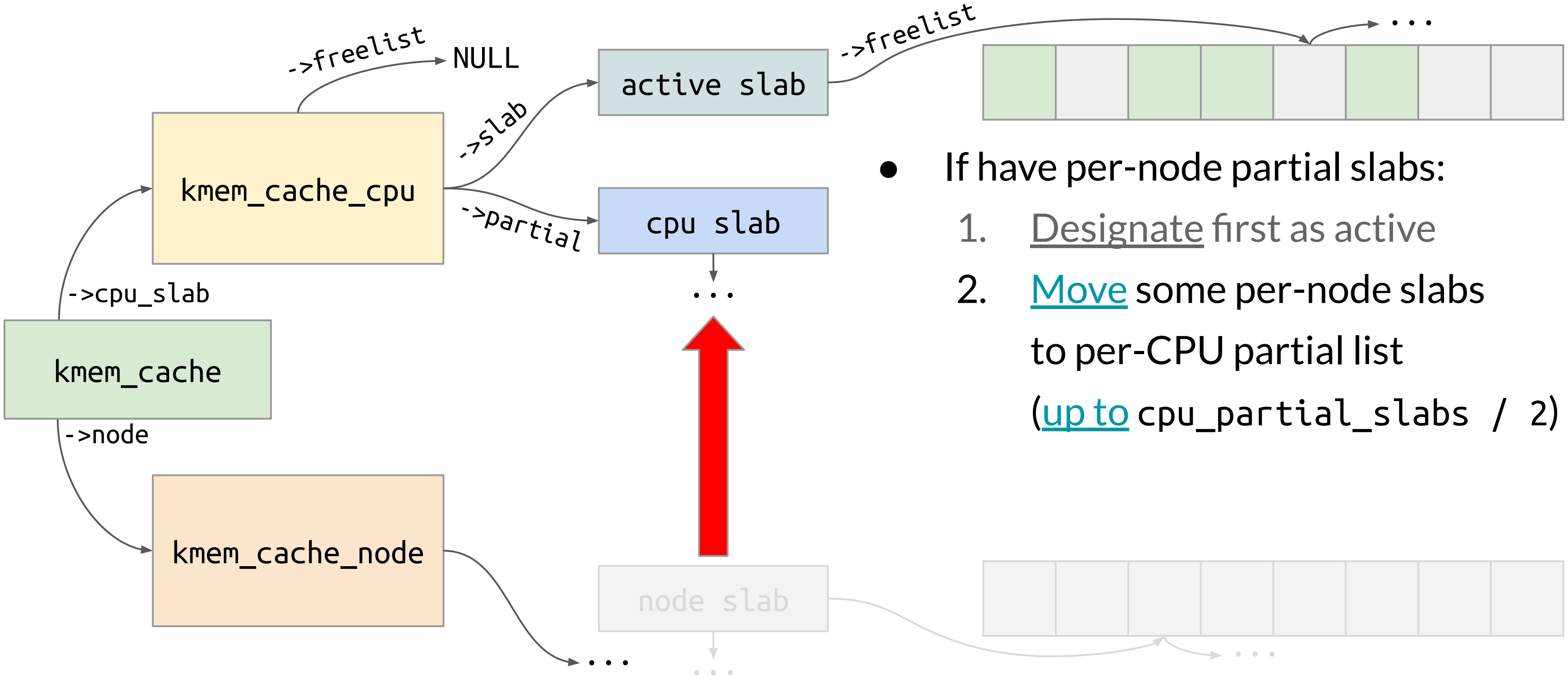
- If have per-node partial slabs

Tier 4: Allocating from per-node partial slabs [1/4]



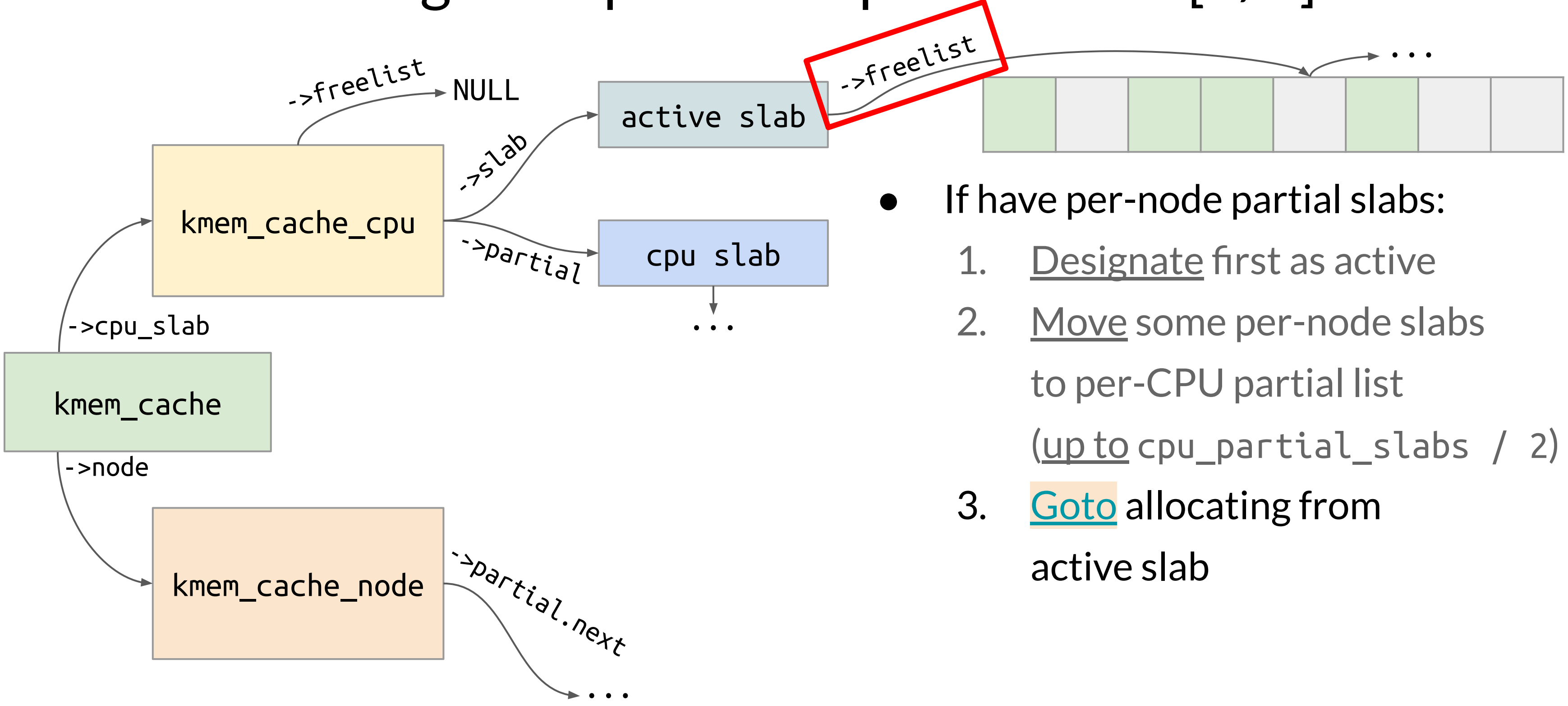
- If have per-node partial slabs:
  1. Designate first as active

Tier 4: Allocating from per-node partial slabs [2/4]



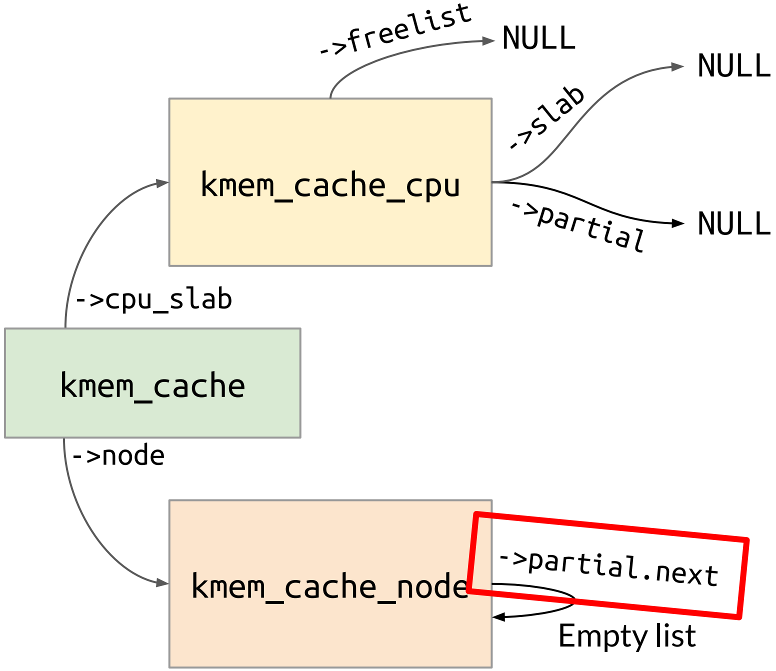
- If have per-node partial slabs:
  1. Designate first as active
  2. Move some per-node slabs to per-CPU partial list (up to `cpu_partial_slabs / 2`)

Tier 4: Allocating from per-node partial slabs [3/4]



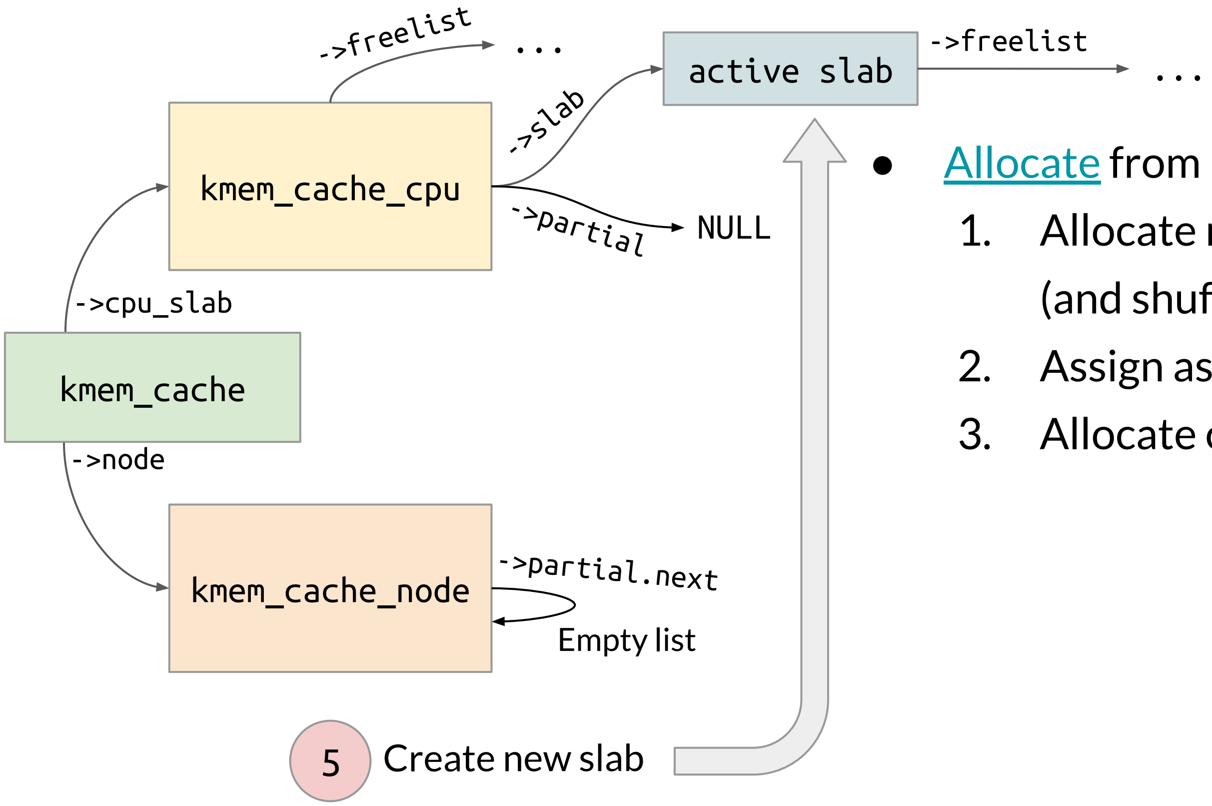
- If have per-node partial slabs:
  1. Designate first as active
  2. Move some per-node slabs to per-CPU partial list (up to `cpu_partial_slabs / 2`)
  3. Goto allocating from active slab

Tier 4: Allocating from per-node partial slabs [4/4]



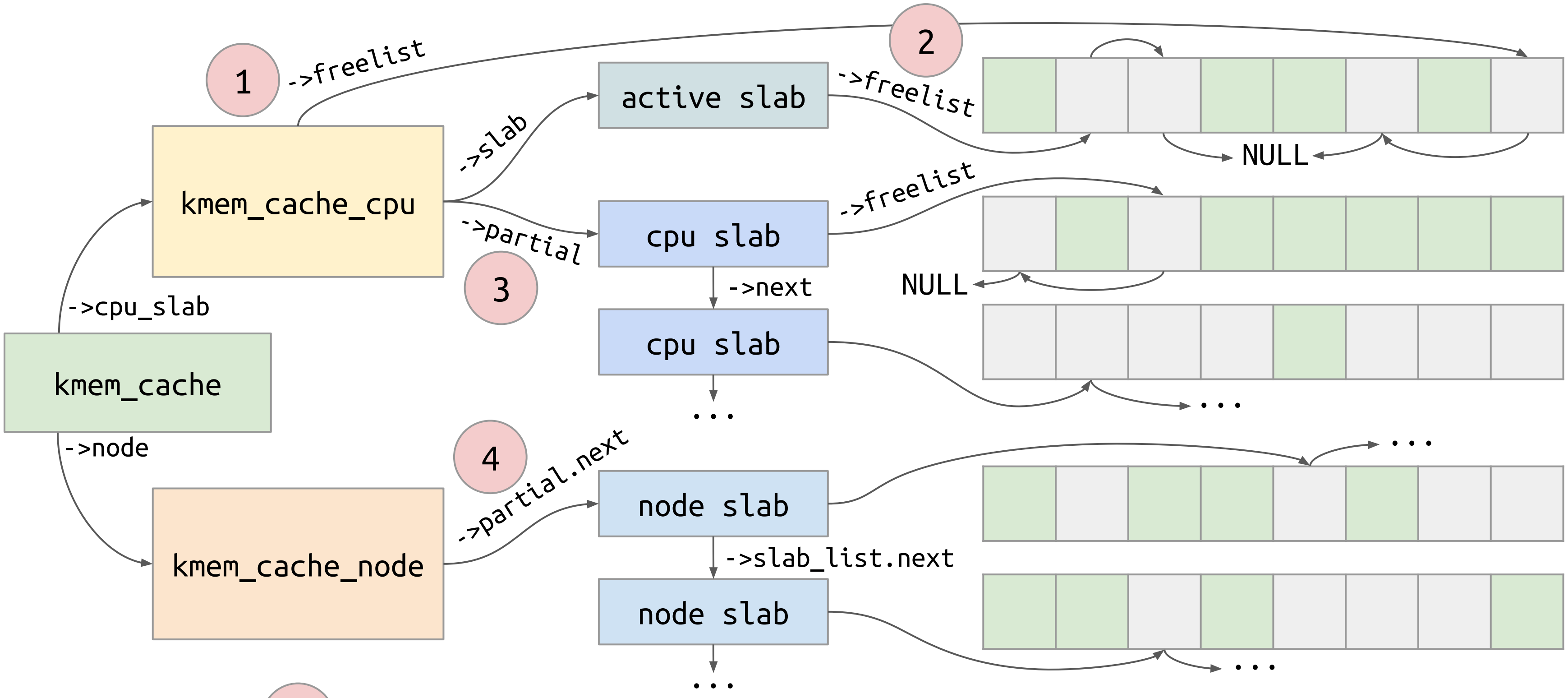
- If per-node partial list is empty, goto Tier 5 (next slide)

Tier 5: Allocating from new slab



- Allocate from new slab:
  1. Allocate new slab from page_alloc (and shuffle its freelist)
  2. Assign as active
  3. Allocate object from this slab

Summary: 5 tiers of allocation process



5 Create new slab

Exploitation-relevant outcomes

1. Each cache has many slabs that store objects
   - With allocated objects from previous kernel operation
   - And with holes — empty object slots

2. Allocations happen from active slab
   - Another slab gets assigned as active once free slots in current one run out

3. Allocating many objects leads to allocation of new empty active slab
   - Once all holes in existing slabs are plugged

Shaping Slab memory:
Out-of-bounds, case #1

Required for attack [1/3]

1. Need kernel bug that leads to OOB access for vulnerable object

- Case #1: Separate allocation and out-of-bounds trigger
  - `IOCTL_ALLOC` — Allocates vulnerable object
  - `IOCTL_OOB` — Writes or reads data out-of-bounds of vulnerable object

Required for attack [2/3]

1. Need kernel bug that leads to OOB access for vulnerable object

2. Need to choose target object to leak or overwrite
    - Assume we chose one

Required for attack [1/3]

1. Need kernel bug that leads to OOB access for vulnerable object

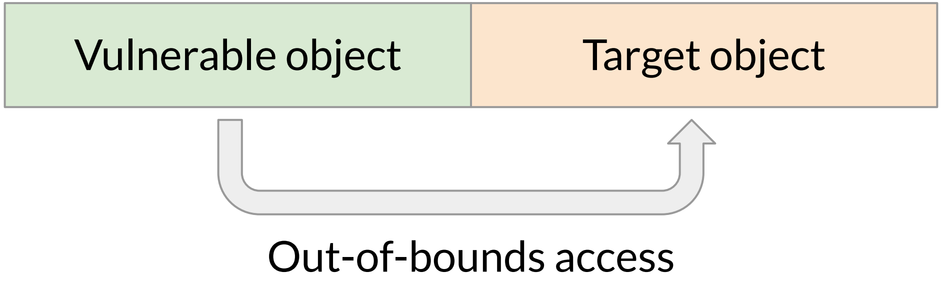
2. Need to choose target object to leak or overwrite

3. Need to shape Slab memory
   - Put vulnerable and target object next to each other

- Focus of this section

Shaping Slab memory for OOB

- How can we achieve the following scenario?
- Cannot just allocate vulnerable and then target, freelist is randomized



- Vulnerable object — particular object affected by found bug
- Target object — object we chose to leak or corrupt

Slab shaping for OOB [0/4]

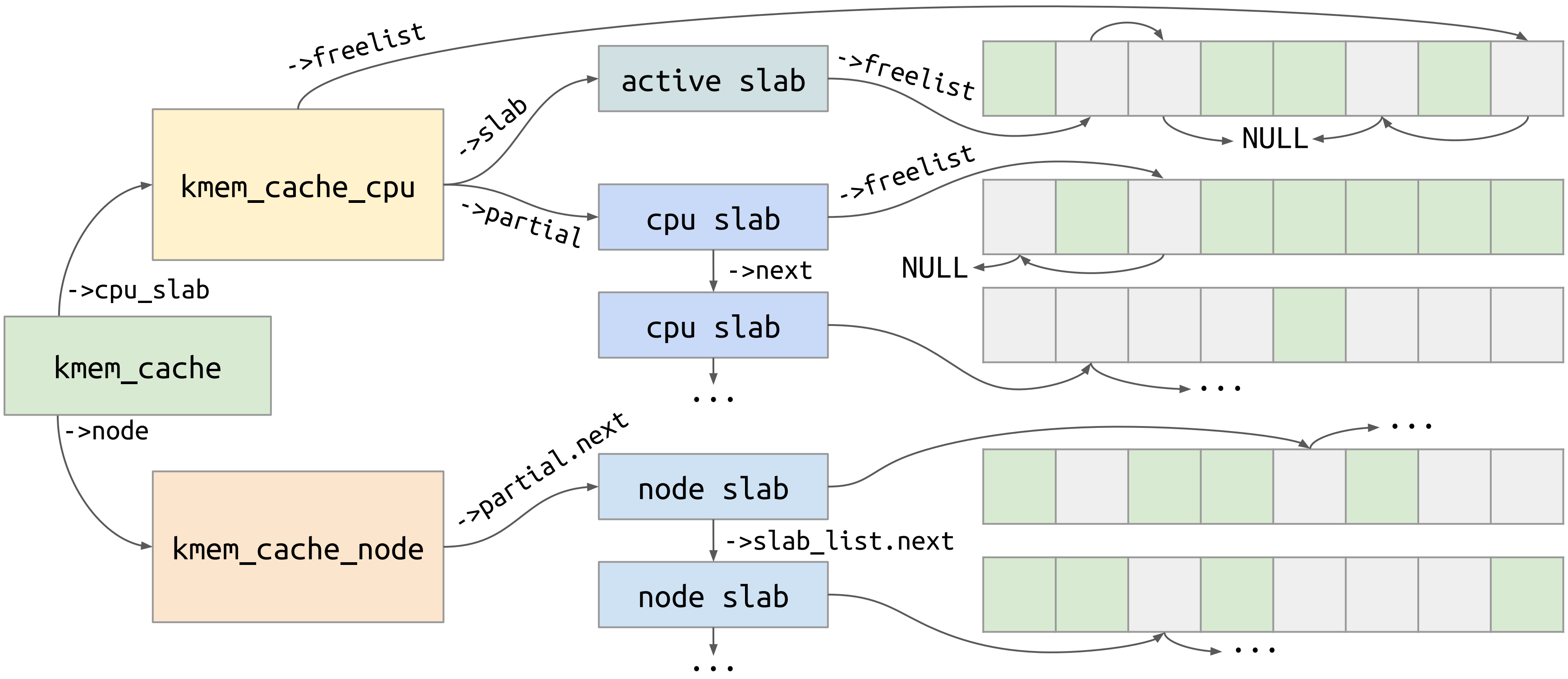
1. Allocate enough target objects to get new active slab
    - Why? Want slab with controlled objects only

      (and don't want other CPUs to mess around)
    - Why target objects? Will discuss later
    - How many to allocate? Let's discuss this

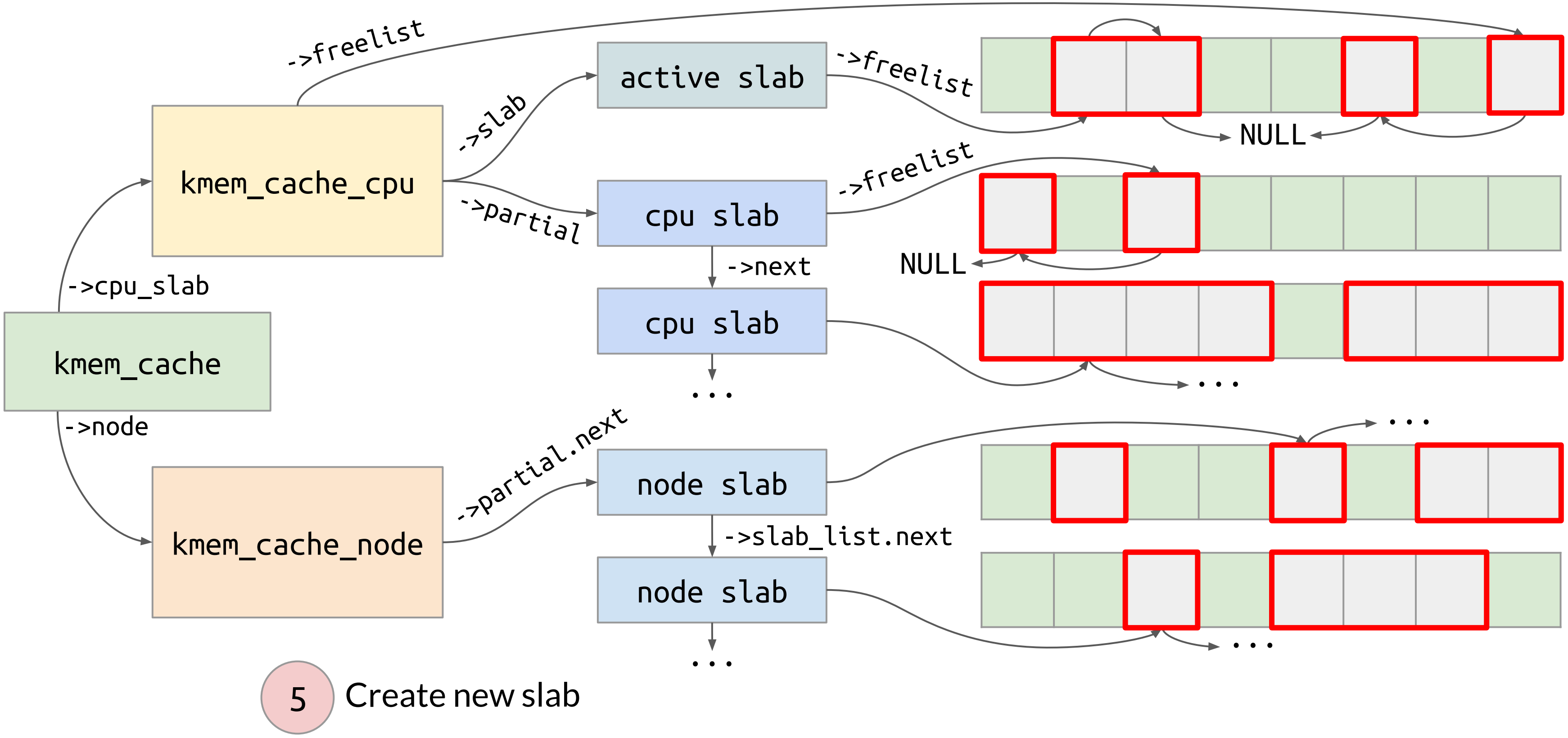
Want new active slab like this:

Target		Target					Target

Slab memory state before shaping



Need to plug all holes (allocate free slots) to reach Tier 5



Finding out number of holes [1/2]

- No way to find out exact number on target system as unprivileged user
  - `/proc/slabinfo` and related files are not user-readable

- No upper limit on total number of holes
  - Active — up to `objperslab` holes
  - Per-CPU partials — up to `objperslab * cpu_partial_slabs` holes
  - Per-node partials — no limit on number of slabs ⇒ no limit on holes

Finding out number of holes [2/2]

- One approach: get an estimate:
  1. Reproduce target environment locally (give it some uptime as well)
  2. Use `/proc/slabinfo` to estimate number of holes (next slides)
  3. Allocate more than the estimate (x2 is a good start)

- Alternative: Rely on timing side channels (more links at the end)
  - Basic idea: system call that allocates new slab takes longer to finish

/proc/slabinfo

# name	<active_objs>	<num_objs>	<objsize>	<objperslab>	<pagesperslab>
cred_jar	7644	7644	192	21	1
kmalloc-8k	456	460	8192	4	8
kmalloc-4k	3118	3160	4096	8	8
kmalloc-2k	3621	3696	2048	16	8
kmalloc-32	54789	55808	32	128	1

- `active_objs` — allocated objects, `num_objs` — total slots in existing slabs
- Numbers are not kept up to date:
  updated when a slab is allocated, freed, or moved to per-node partial list

Forcing `/proc/slabinfo` update

```
# name          <active_objs> <num_objs>
kmalloc-32            25216      25216     // Before shrinking.
kmalloc-32            23132      24320     // After shrinking.
```

- Shrink cache to get more accurate `active_objs` and `num_objs`:

  `echo 1 | sudo tee /sys/kernel/slab/kmalloc-32/shrink`

- After shrinking, number of holes == `num_objs - active_objs`
  - Shrinking frees empty slabs from partial lists ⇒ Number of holes changes
  - But number is inaccurate anyway, as we reproduce environment

Slab shaping for OOB [1/4]

1. Allocate enough target objects to get new active slab
    - Result: New active slab partially-filled with target objects

New active slab:

Target		Target					Target

Slab shaping for OOB [2/4]

1. Allocate enough target objects to get new active slab
2. Allocate one vulnerable object via `IOCTL_ALLOC`
    - Vulnerable object is allocated in active slab

New active slab with vulnerable object:

Target		Target	Vuln				Target

Slab shaping for OOB [3/4]

1. Allocate enough target objects to get new active slab
2. Allocate one vulnerable object via `IOCTL_ALLOC`
3. Allocate enough target objects to fill active slab
   - `objperslab - 1` is enough
   - Slab gets full and thus stops being active, but we don't care

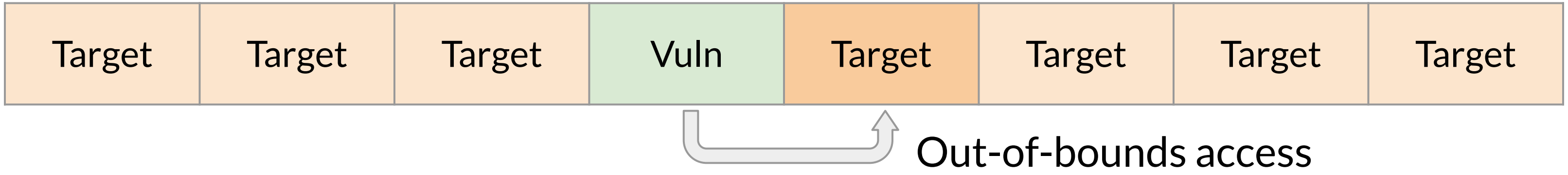
Vulnerable object is now followed by target object:

Target	Target	Target	Vuln	Target	Target	Target	Target

Slab shaping for OOB [4/4]

1. Allocate enough target objects to get new active slab
2. Allocate one vulnerable object via `IOCTL_ALLOC`
3. Allocate enough target objects to fill active slab
4. Trigger out-of-bounds access via `IOCTL_OOB`

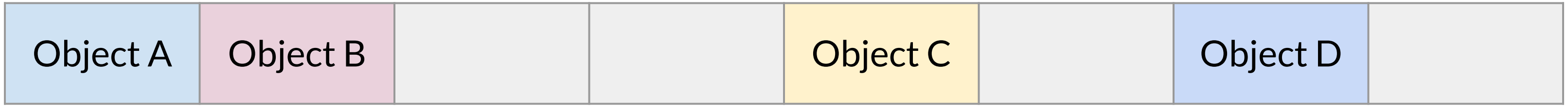
Out-of-bounds from vulnerable object lands in target object:



What if we skip step #1? [1/2]

1. ~~Allocate enough target objects to get new active slab~~

Original active slab:

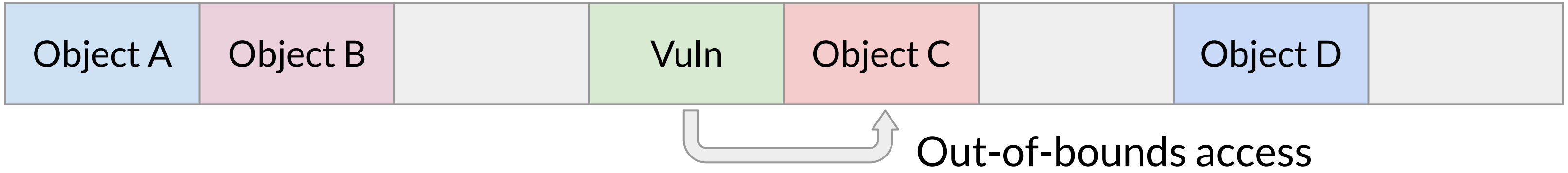


What if we skip step #1? [2/2]

1. ~~Allocate enough target objects to get new active slab~~

- ⇒ Target object might land before slot taken by unknown object
    - ⇒ Will corrupt important memory instead of target

Original active slab with vulnerable object:



What if we use different objects for step #1? [1/3]

1. Allocate enough plug objects to get new active slab

New active slab:

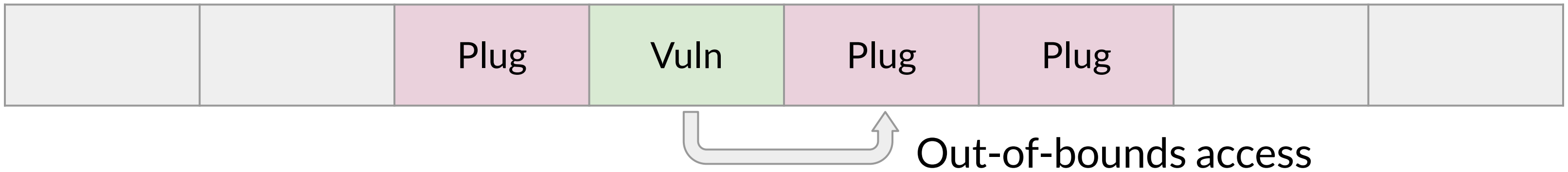
		Plug		Plug	Plug		

What if we use different objects for step #1? [2/3]

1. Allocate enough plug objects to get new active slab

- ⇒ Target object might land before plug object
  - ⇒ Will fail to overwrite target

New active slab with vulnerable object:

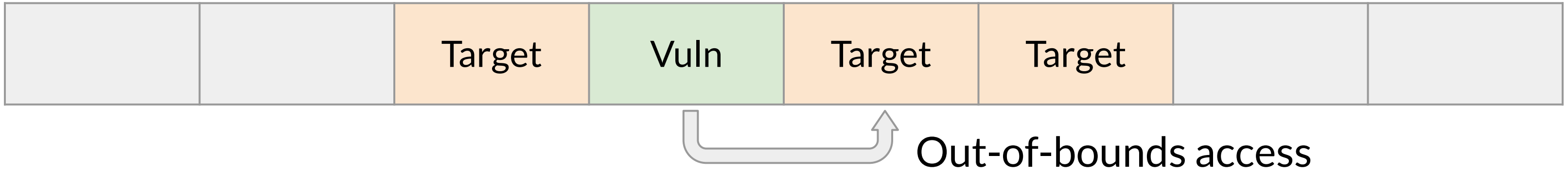


What if we use different objects for step #1? [3/3]

1. Allocate enough ~~plug~~ target objects to get new active slab

- Using target objects for step #1 solves this problem
  - (Technically, can plug holes via plug objects
    and then allocate enough target objects to get new slab)

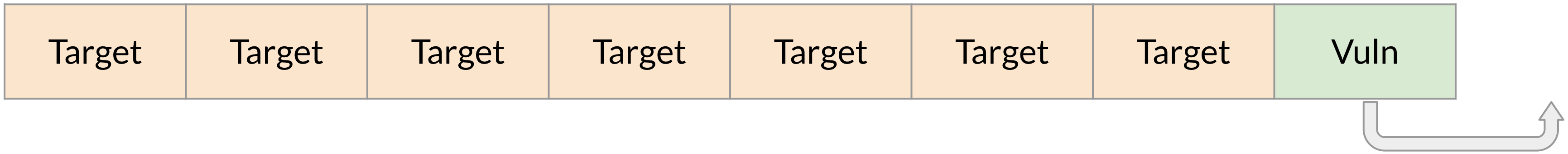
New active slab with vulnerable object:



Things can go wrong #1: Last object

- Vulnerable object might be last object in slab
  - ⇒ No target object follows, Slab shaping fails

- ⇒ Slab shaping is best-effort
  - Unless another slab with target objects follows (requires page_alloc shaping, out-of-scope)



Things can go wrong #2: Migration

- Migration — process being moved to another CPU

- If exploit is migrated to another CPU during execution, Slab shaping breaks
  - Objects are allocated from active slab for current CPU
  - After migration, exploits switches to different active slab

CPU pinning to deal with migration

- Solution: Pin exploit process to one CPU via `sched_setaffinity`
  - Can be done by unprivileged user, inherited by forked processes

```
cpu_set_t my_set;
CPU_ZERO(&my_set);
CPU_SET(0, &my_set);
sched_setaffinity(0, sizeof(my_set), &my_set)
```

Things can go wrong #3: Preemption

- Preemption — another task or interrupt handler getting scheduled on the same CPU instead of running process
  - CPU pinning does not prevent preemption

- If exploit is preempted, Slab shaping might fail
  - Another task might allocate or free objects from/to the same cache

Ideas for dealing with preemption

- Minimize time for Slab shaping: e.g., don't sleep/print during shaping
- If have choice, use less noisy (less frequently used) cache
- Get fresh scheduler time slice via `sched_yield` before shaping
- Check if exploit got preempted via `/proc/self/status`
- Rely on timing side channels (more links at the end)

- But no perfect solution
  - ⇒ Slab shaping is best-effort

Shaping Slab memory: Out-of-bounds, case #2

Case #2: Combined allocation and out-of-bounds trigger

- Before:
  - `IOCTL_ALLOC` — Allocates vulnerable object
  - `IOCTL_OOB` — Writes or reads data out-of-bounds of vulnerable object

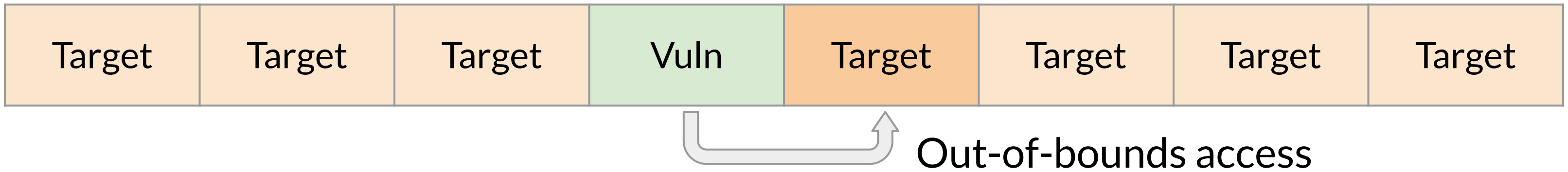
- Now:
  - `IOCTL_ALLOC_AND_OOB` — Allocates vulnerable object and immediately writes data out-of-bounds

Reminder: Slab shaping for OOB

1. Allocate enough target objects to get new active slab
2. Allocate one vulnerable object via `IOCTL_ALLOC`
3. Allocate enough target objects to fill active slab
4. Trigger out-of-bounds access via `IOCTL_OOB`

With combined OOB,
cannot separate steps 2 and 4

Out-of-bounds from vulnerable object lands in target object:



Allocation-only approach for combined OOB [1/4]

1. Allocate enough target objects to get new active slab

Another approach discussed later

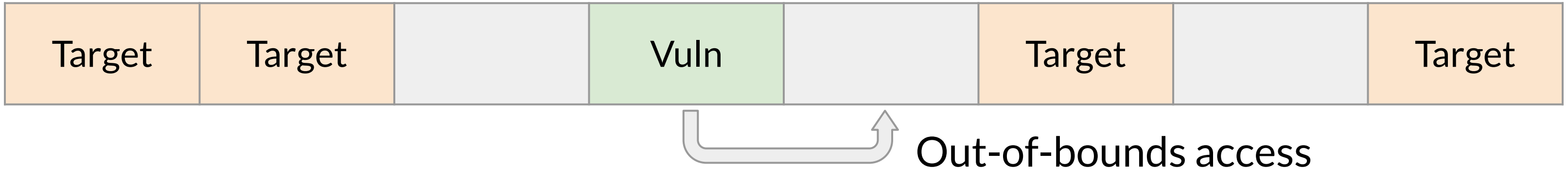
New active slab:

Target	Target				Target		Target

Allocation-only approach for combined OOB [2/4]

1. Allocate enough target objects to get new active slab
2. Allocate one vulnerable object and trigger OOB via `IOCTL_ALLOC_AND_OOB`

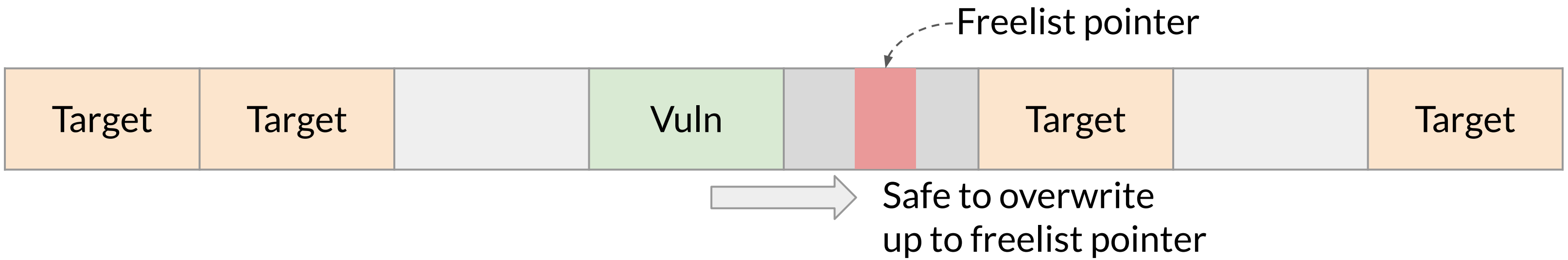
New active slab with vulnerable object:



Allocation-only approach for combined OOB [3/4]

1. Allocate enough target objects to get new active slab
2. Allocate one vulnerable object and trigger OOB via `IOCTL_ALLOC_AND_OOB`

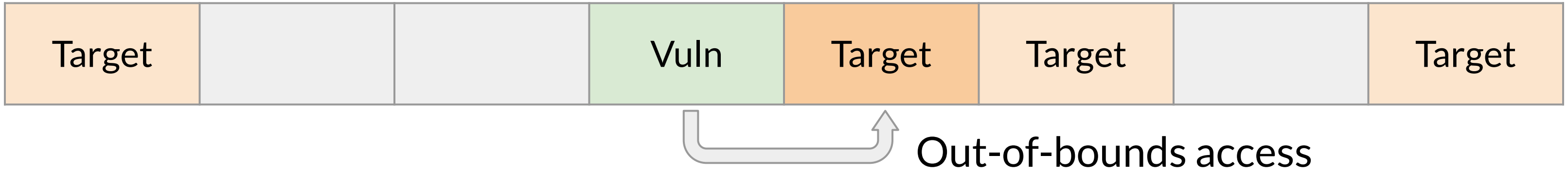
- Case #1: Out-of-bounds access lands in free slot
  - As long as freelist pointer is not corrupted (OOB size is small), nothing happens ⇒ Can retry shaping and OOB triggering



Allocation-only approach for combined OOB [4/4]

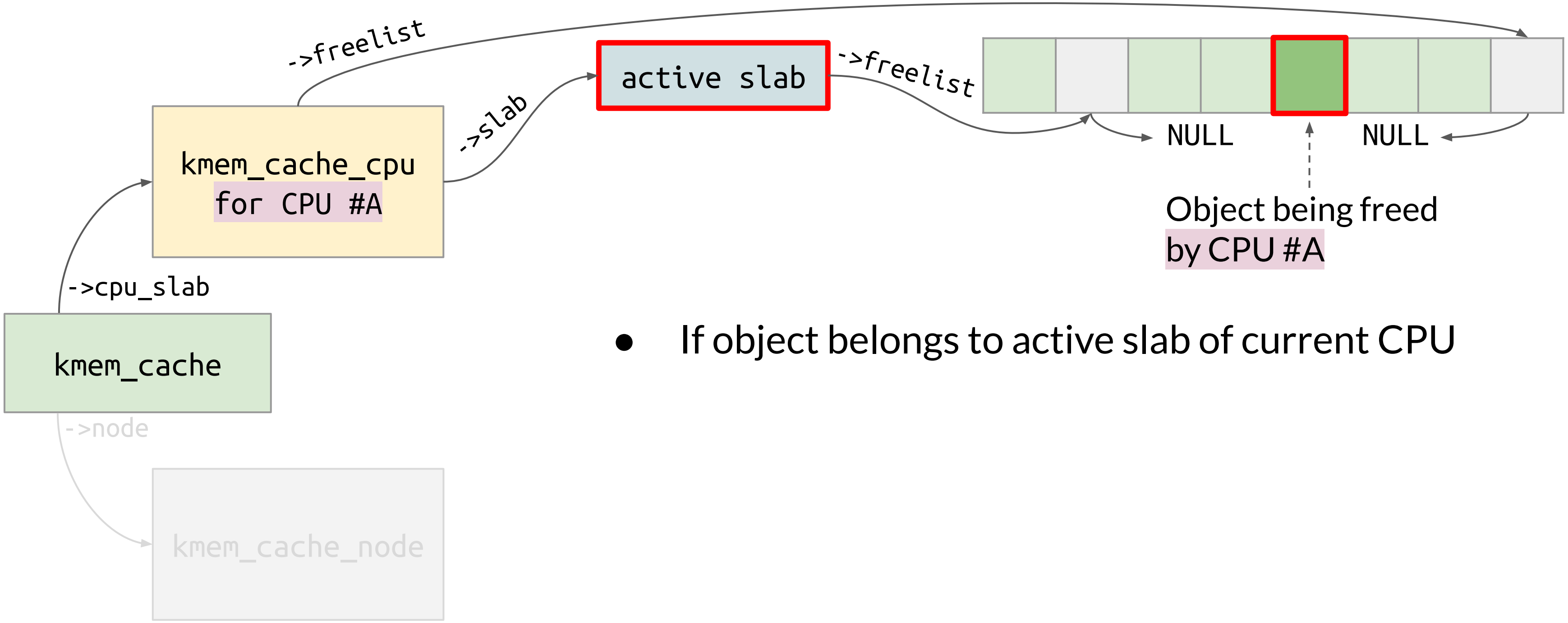
1. Allocate enough target objects to get new active slab
2. Allocate one vulnerable object and trigger OOB via `IOCTL_ALLOC_AND_OOB`

- Case #2: Out-of-bounds access lands in target object
    - Success: target object overwritten
    - But might need multiple retries to achieve this



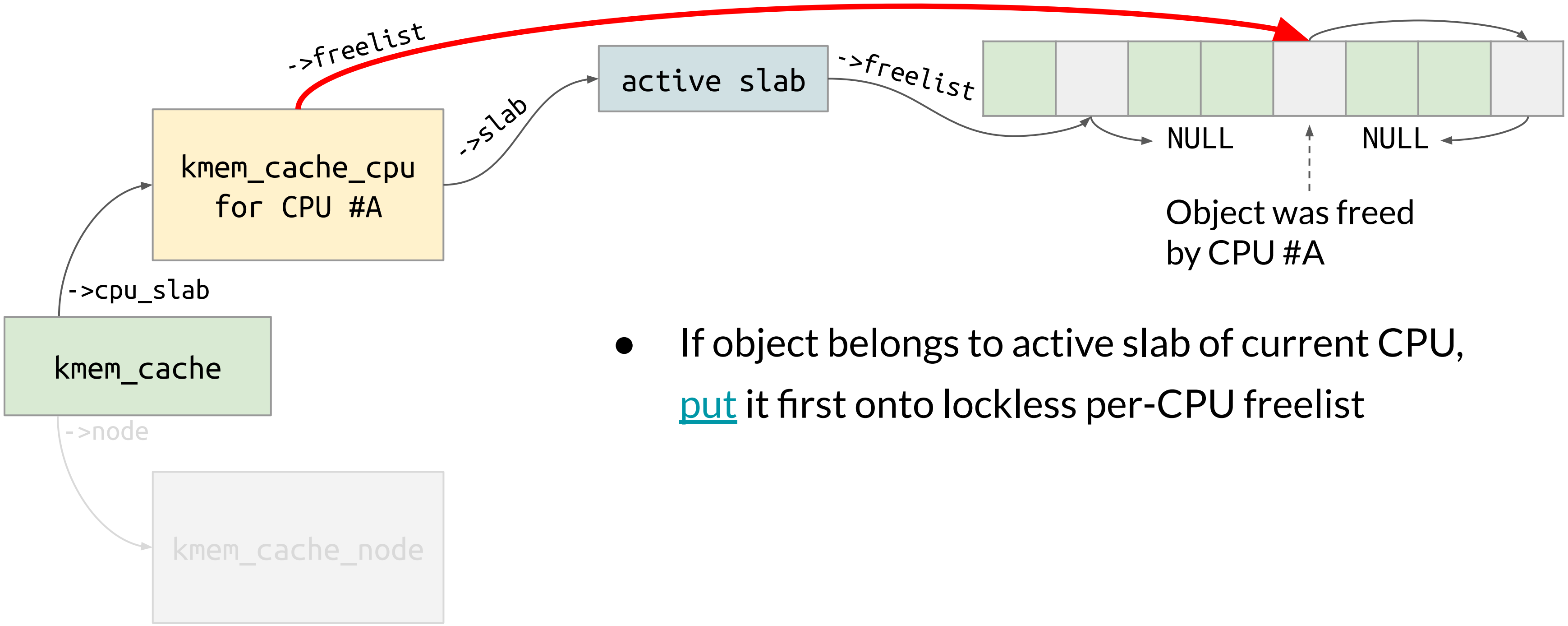
SLUB internals:
Freeing process, part #1

Case #1: Object belongs to active slab of current CPU [0/1]



- If object belongs to active slab of current CPU

Case #1: Object belongs to active slab of current CPU [1/1]



- If object belongs to active slab of current CPU, put it first onto lockless per-CPU freelist

Exploitation-relevant outcome

- Last freed object is first to be allocated if it belongs to active slab

- Imagine scenario:

```
void *ptr1 = kmalloc(128, GFP_KERNEL);
free(ptr1);
void *ptr2 = kmalloc(128, GFP_KERNEL);
free(ptr2);
void *ptr3 = kmalloc(128, GFP_KERNEL);
```

- `ptr1`, `ptr2`, and `ptr3` all point to the same object bouncing back and forth from/to lockless per-CPU freelist of active slab for `kmalloc-128`

Shaping Slab memory: Use-after-free, case #1

Case #1: Use-after-free on active slab

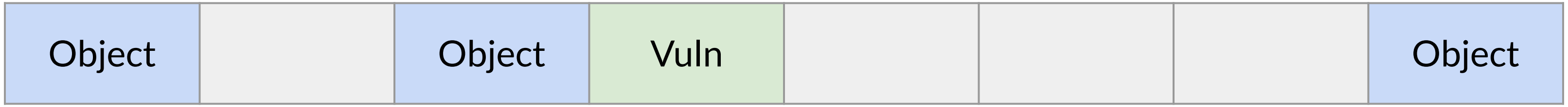
- Assume we have use-after-free vulnerability:
  - `IOCTL_ALLOC` — Allocates vulnerable object
  - `IOCTL_FREE` — Frees vulnerable object
  - `IOCTL_UAF` — Writes or reads data of vulnerable object,
    can be used after `IOCTL_FREE`

- Want to shape slab memory to:
  - Put target object into slot of freed vulnerable object
  - And trigger use-after-free

Slab shaping for UAF on active slab [1/4]

1. Allocate one vulnerable object via `IOCTL_ALLOC`
    - Object allocated from active slab
    - No need to plug holes for this approach

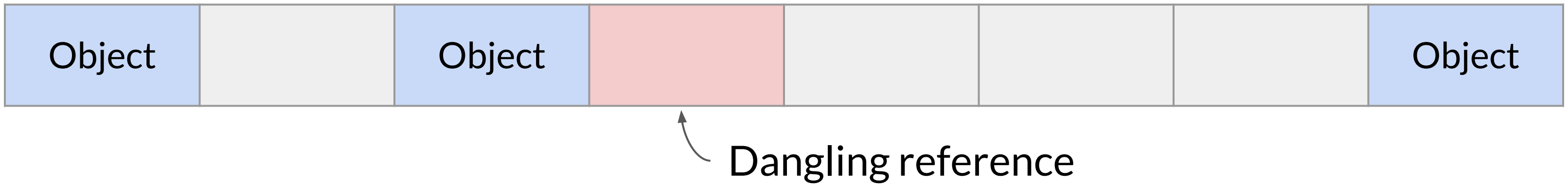
Active slab with vulnerable object:



Slab shaping for UAF on active slab [2/4]

1. Allocate one vulnerable object via `IOCTL_ALLOC`
2. Free vulnerable object via `IOCTL_FREE`, dangling reference remains
    - Slot placed first onto lockless per-CPU freelist of active slab

Active slab with vulnerable object freed:



Slab shaping for UAF on active slab [3/4]

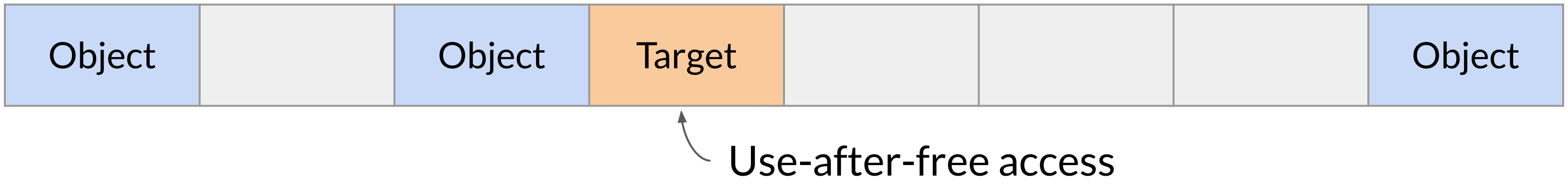
1. Allocate one vulnerable object via `IOCTL_ALLOC`
2. Free vulnerable object via `IOCTL_FREE`, dangling reference remains
3. Allocate one target object, dangling reference points to it
    - Object allocated from lockless per-CPU freelist ⇒ Takes up the same slot

Active slab with vulnerable object replaced:

Slab shaping for UAF on active slab [4/4]

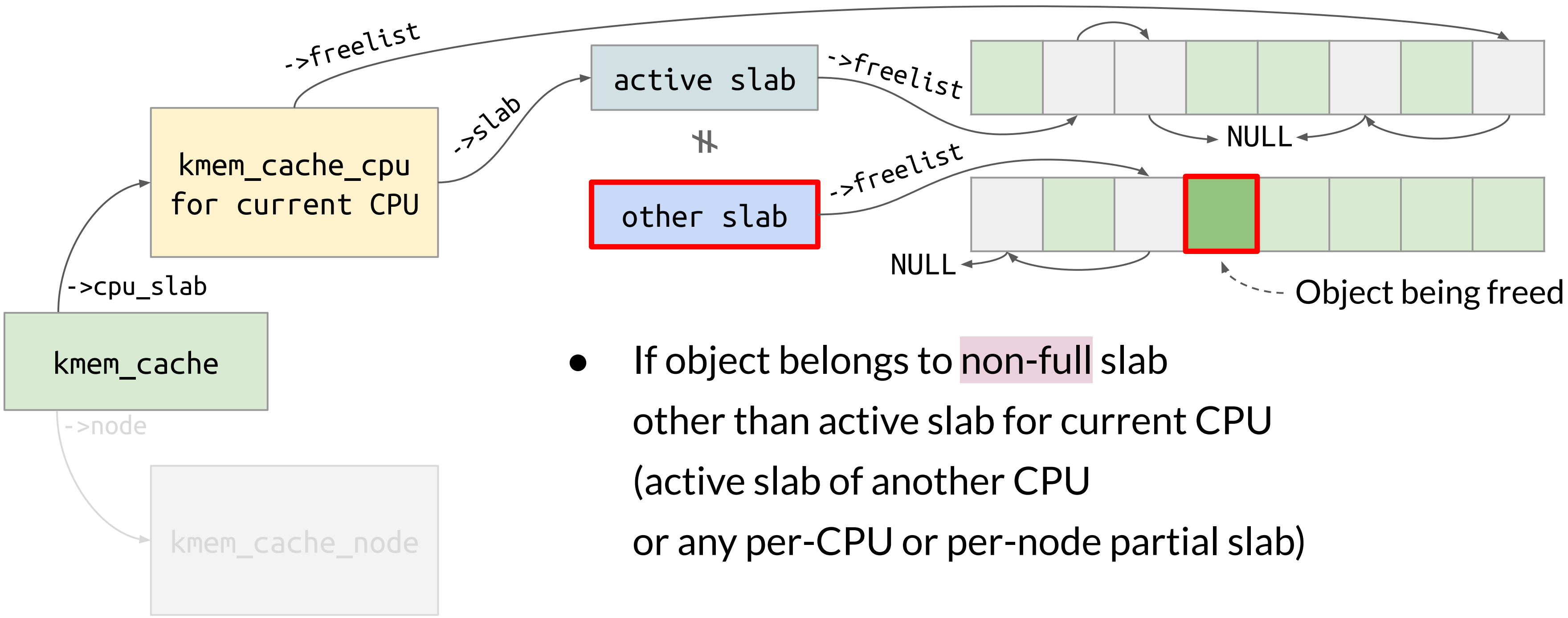
1. Allocate one vulnerable object via `IOCTL_ALLOC`
2. Free vulnerable object via `IOCTL_FREE`, dangling reference remains
3. Allocate one target object, dangling reference points to it
4. Now can trigger use-after-free access via `IOCTL_UAF`

Use-after-free access lands in target object:



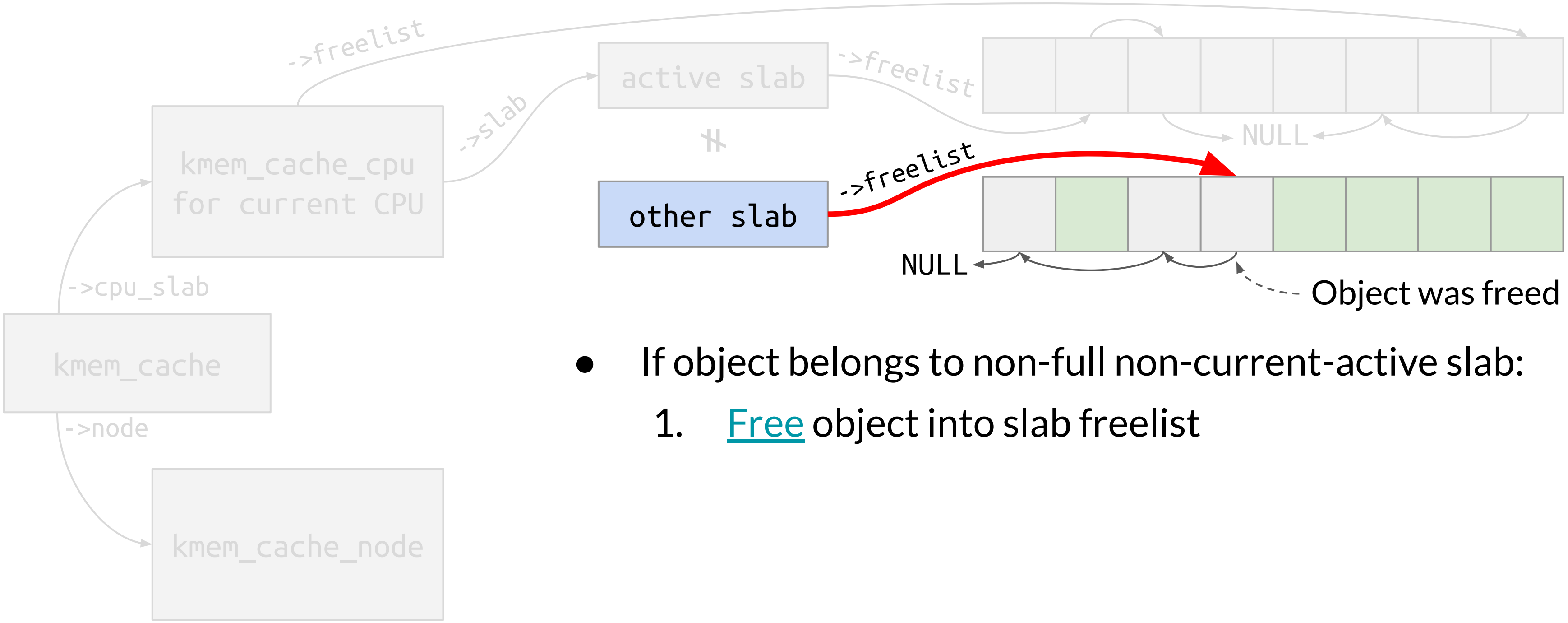
SLUB internals: Freeing process, part #2

Case #2: Object belongs to another non-full slab [0/4]



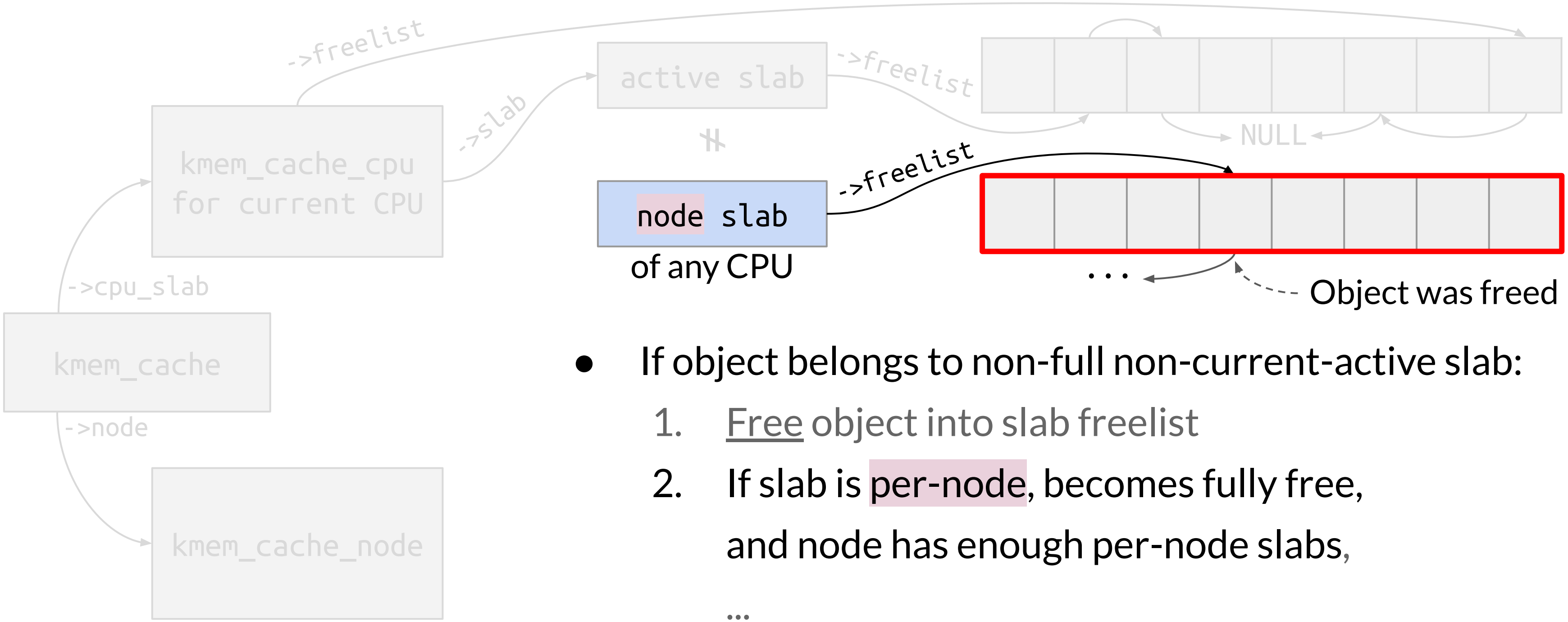
- If object belongs to non-full slab
  other than active slab for current CPU
  (active slab of another CPU
  or any per-CPU or per-node partial slab)

Case #2: Object belongs to another non-full slab [1/4]



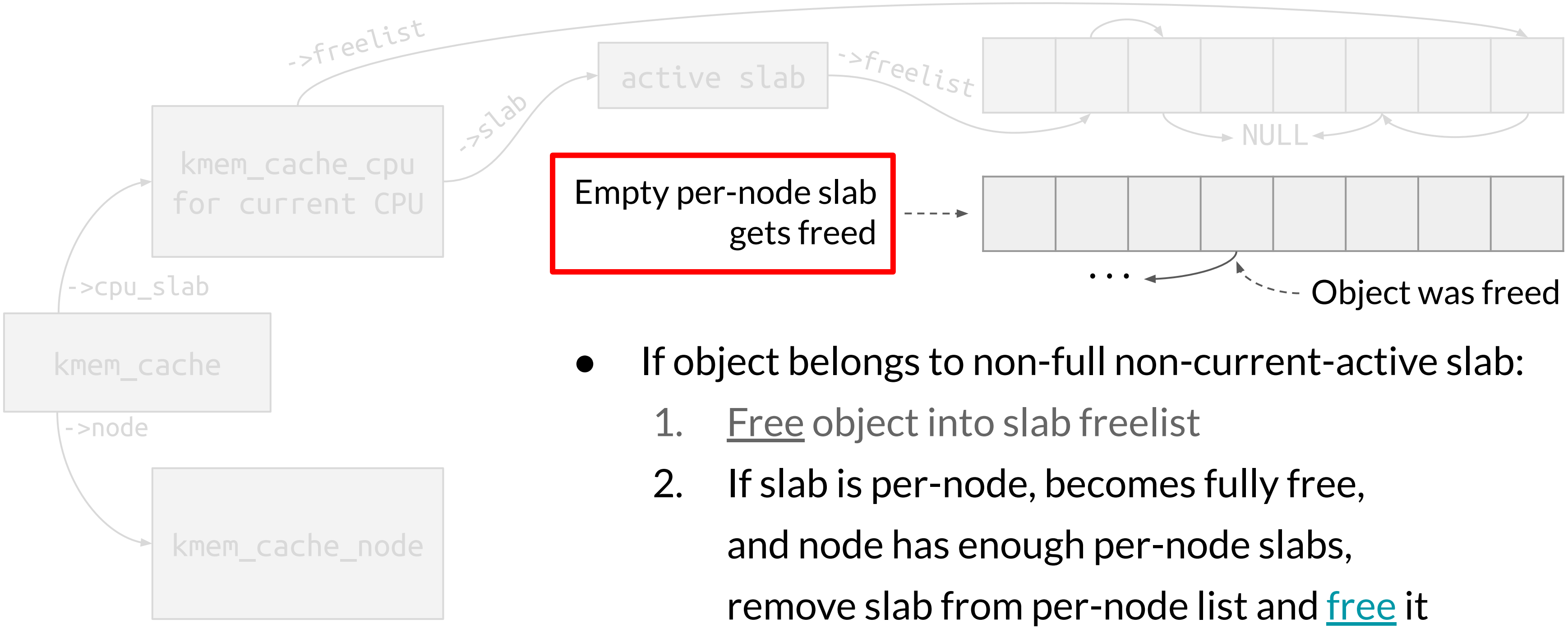
- If object belongs to non-full non-current-active slab:
  1. Free object into slab freelist

Case #2: Object belongs to another non-full slab [2/4]



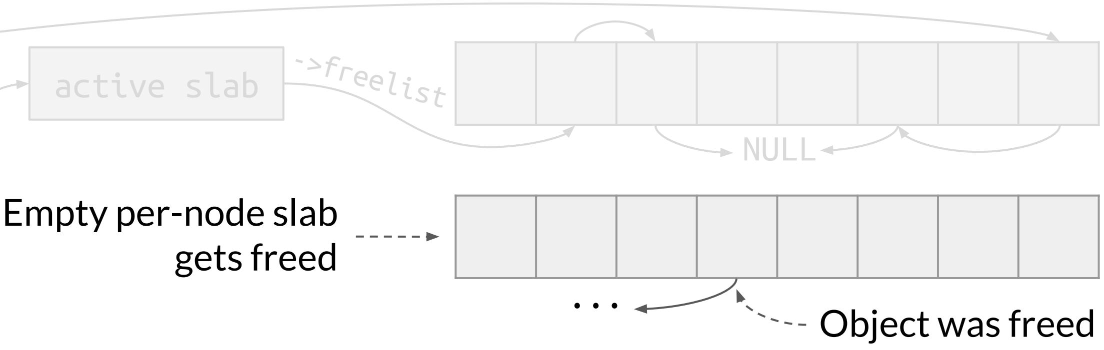
- If object belongs to non-full non-current-active slab:
    1. Free object into slab freelist
    2. If slab is per-node, becomes fully free,
       and node has enough per-node slabs,
       ...

Case #2: Object belongs to another non-full slab [3/4]



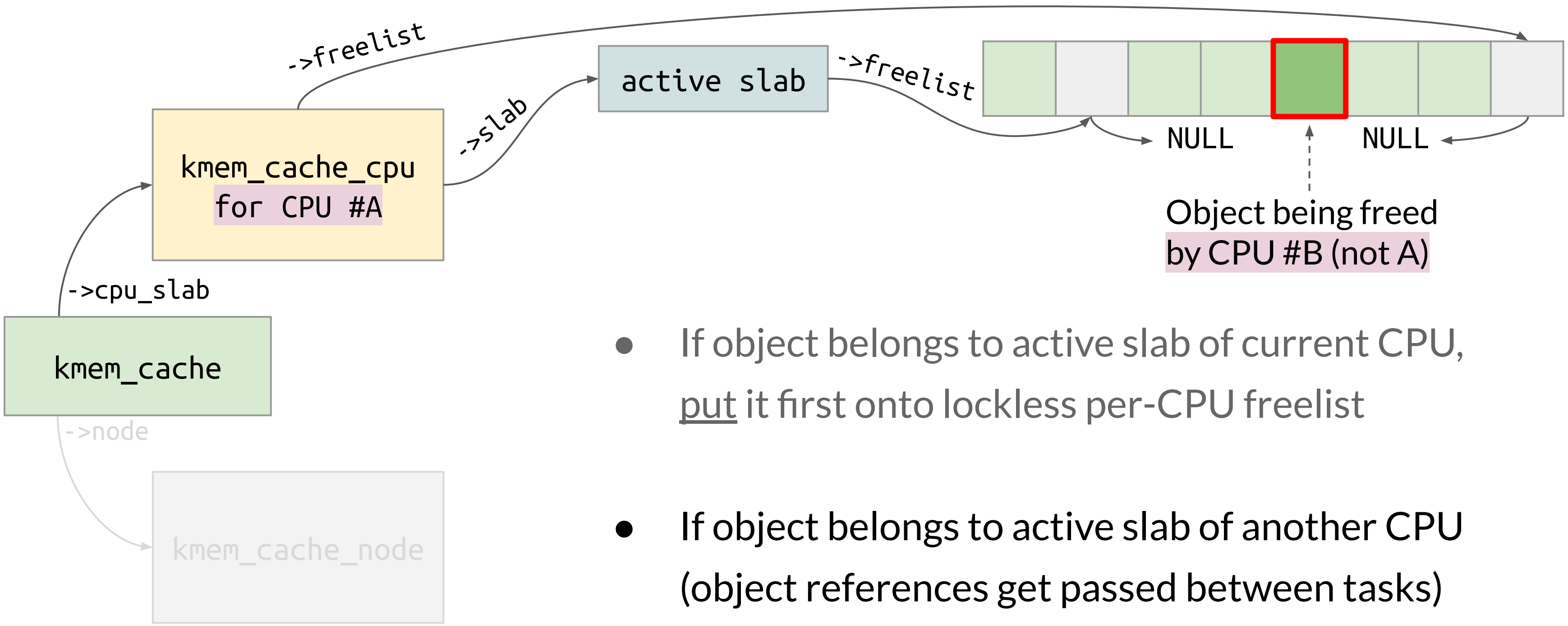
- If object belongs to non-full non-current-active slab:
    1. Free object into slab freelist
    2. If slab is per-node, becomes fully free, and node has enough per-node slabs, remove slab from per-node list and free it

Case #2: Object belongs to another non-full slab [4/4]



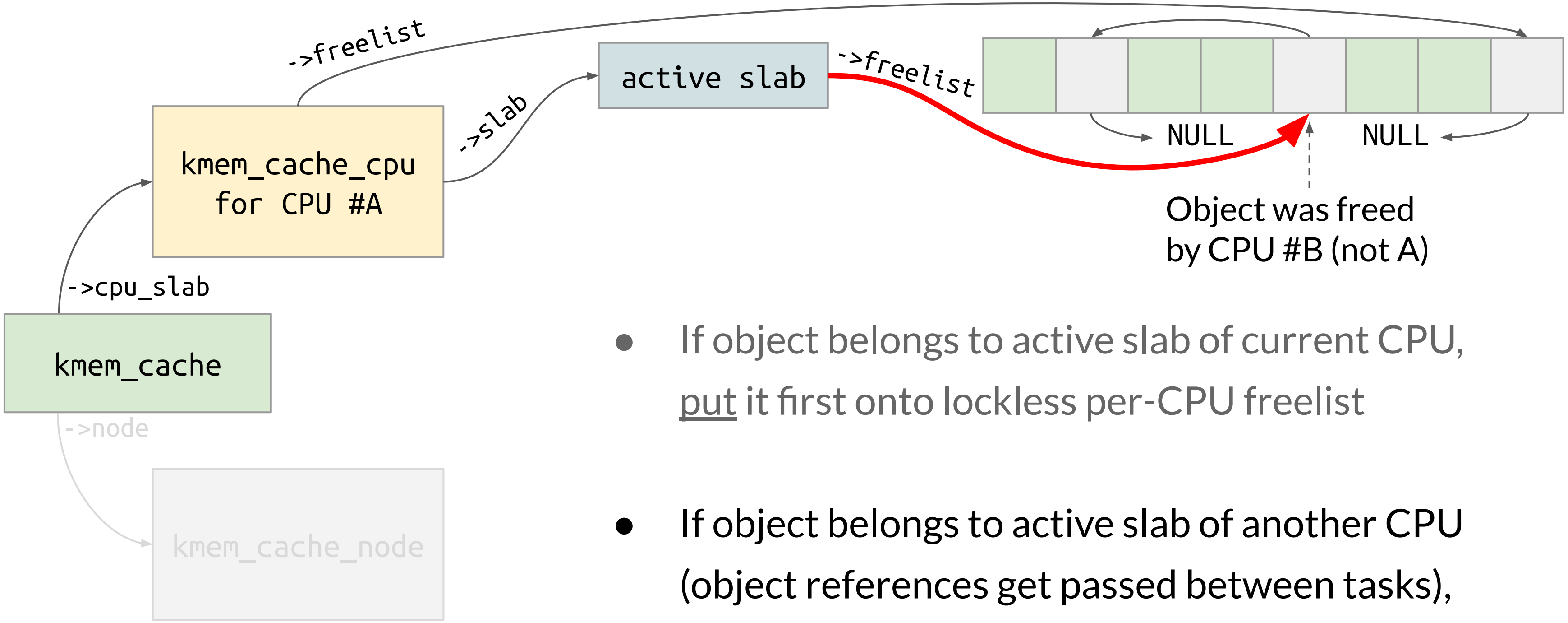
- If object belongs to non-full non-current-active slab:
  1. Free object into slab freelist
  2. Possibly free empty per-node slab
     (does not apply to per-CPU or active slabs)

Example: Object belongs to active slab of another CPU [0/1]



- If object belongs to active slab of current CPU, put it first onto lockless per-CPU freelist
- If object belongs to active slab of another CPU (object references get passed between tasks)

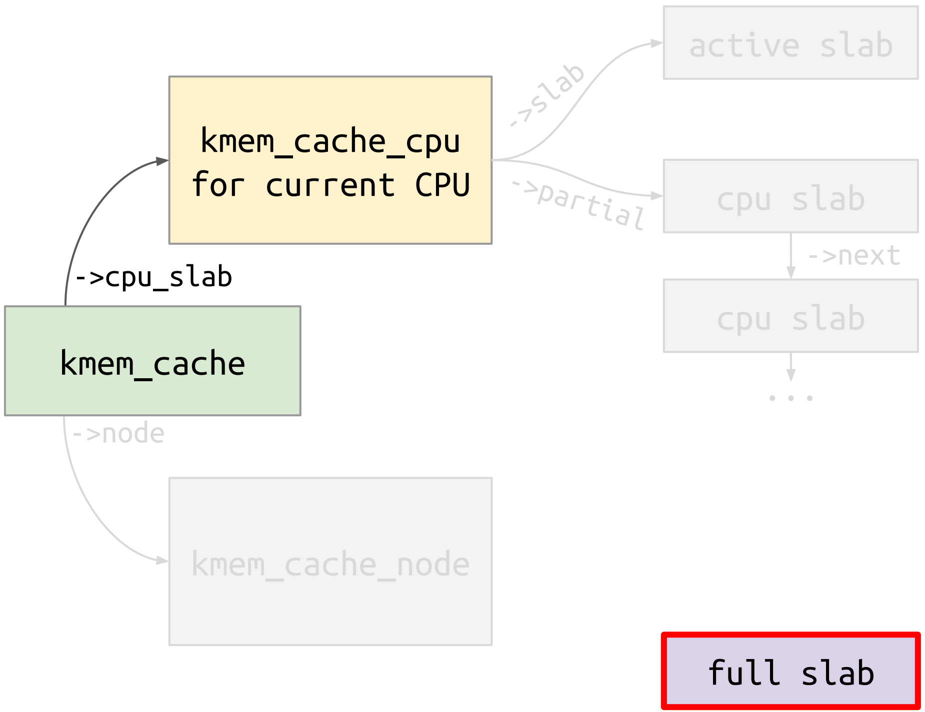
Example: Object belongs to active slab of another CPU [1/1]



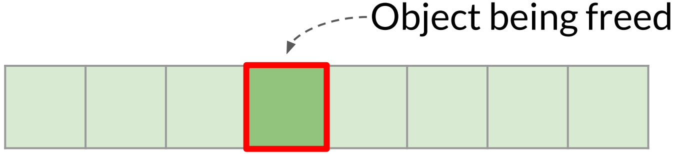
- If object belongs to active slab of current CPU, put it first onto lockless per-CPU freelist

- If object belongs to active slab of another CPU (object references get passed between tasks), put it first onto slab freelist of active slab

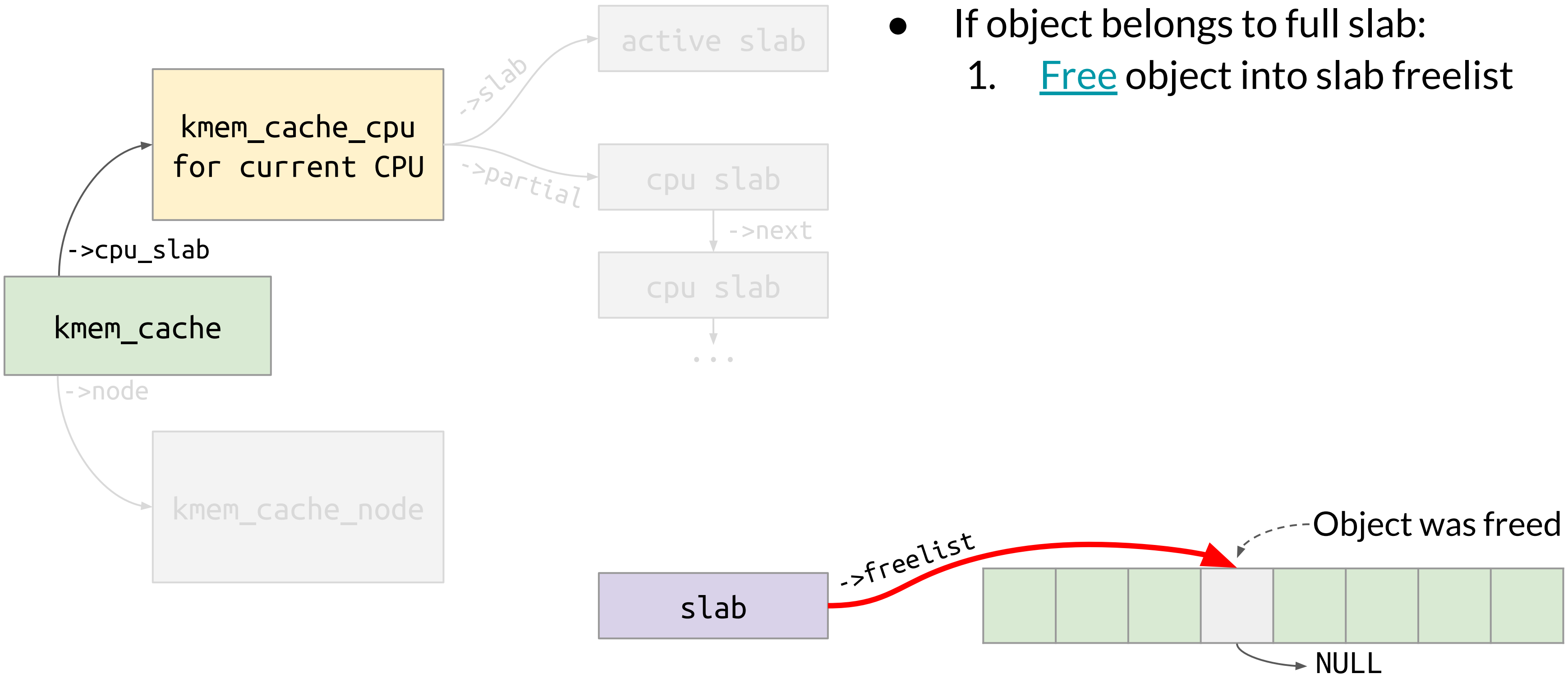
Case #3: Object belongs to full slab [0/2]



- If object belongs to full slab

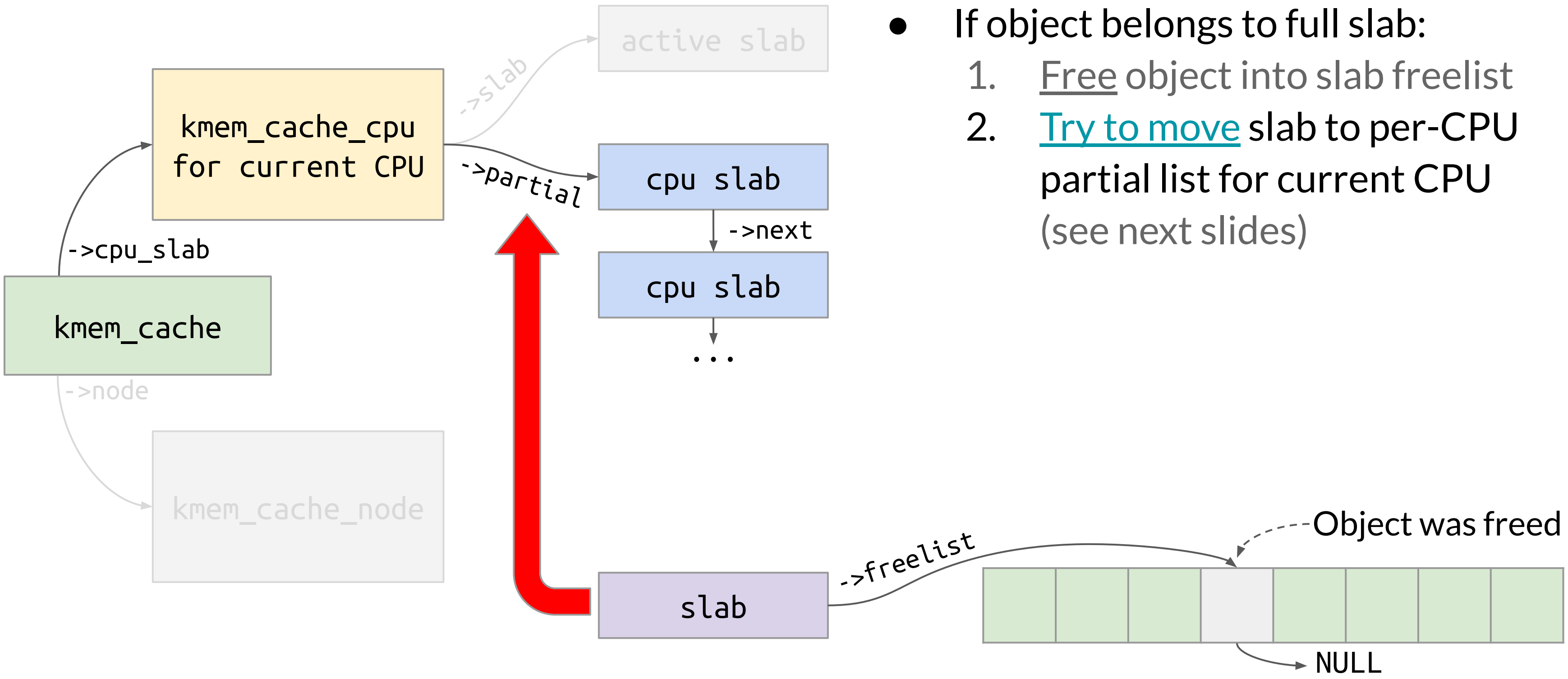


Case #3: Object belongs to full slab [1/2]



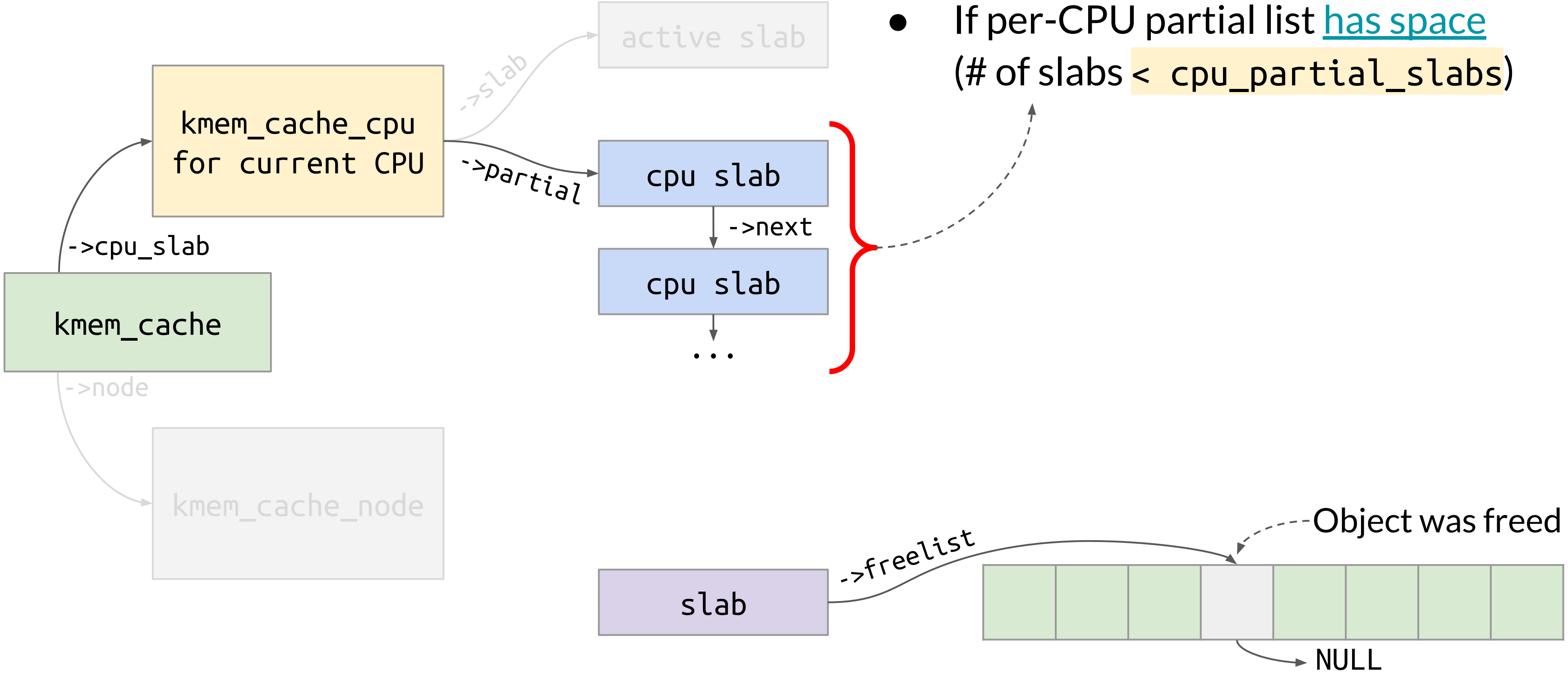
- If object belongs to full slab:
    1. Free object into slab freelist

Case #3: Object belongs to full slab [2/2]



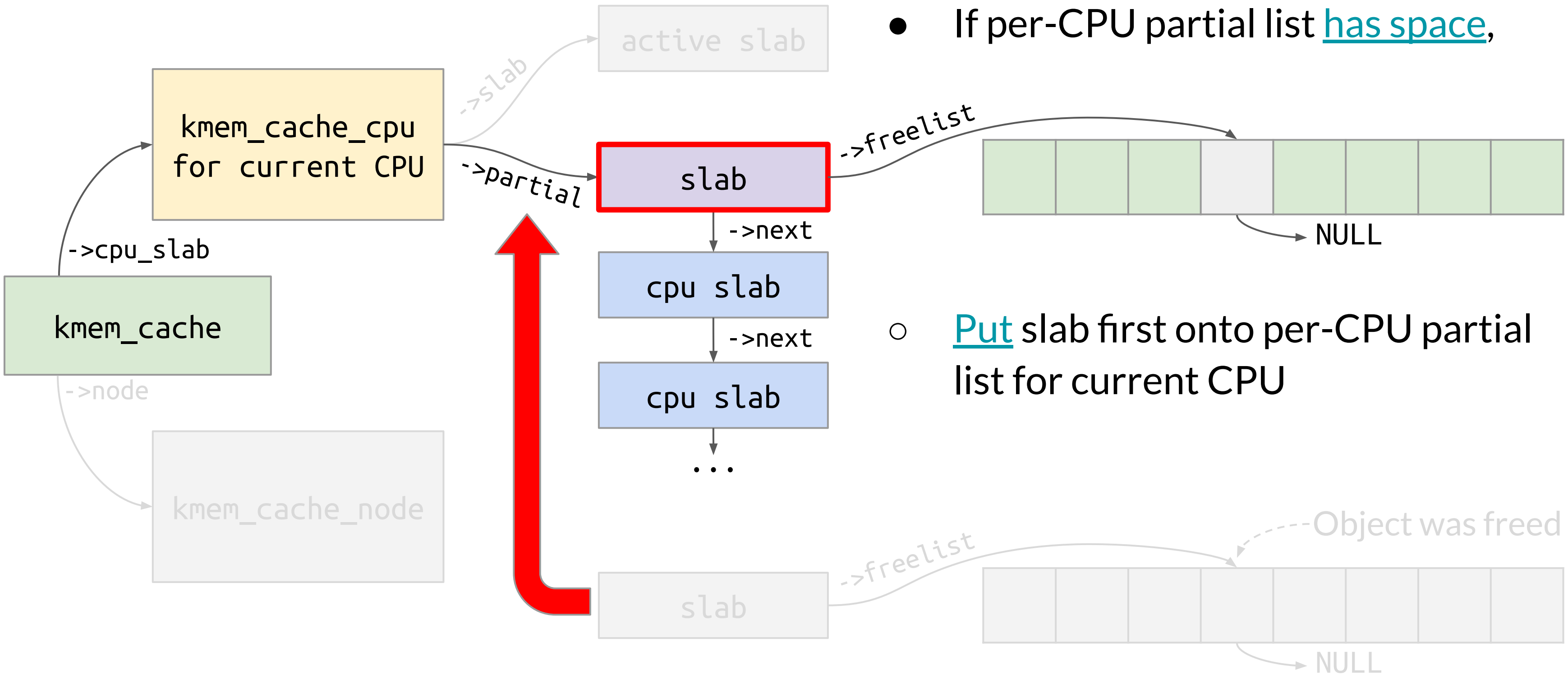
- If object belongs to full slab:
  1. Free object into slab freelist
  2. Try to move slab to per-CPU partial list for current CPU (see next slides)

Subcase #1: Per-CPU partial list does not overflow [0/1]



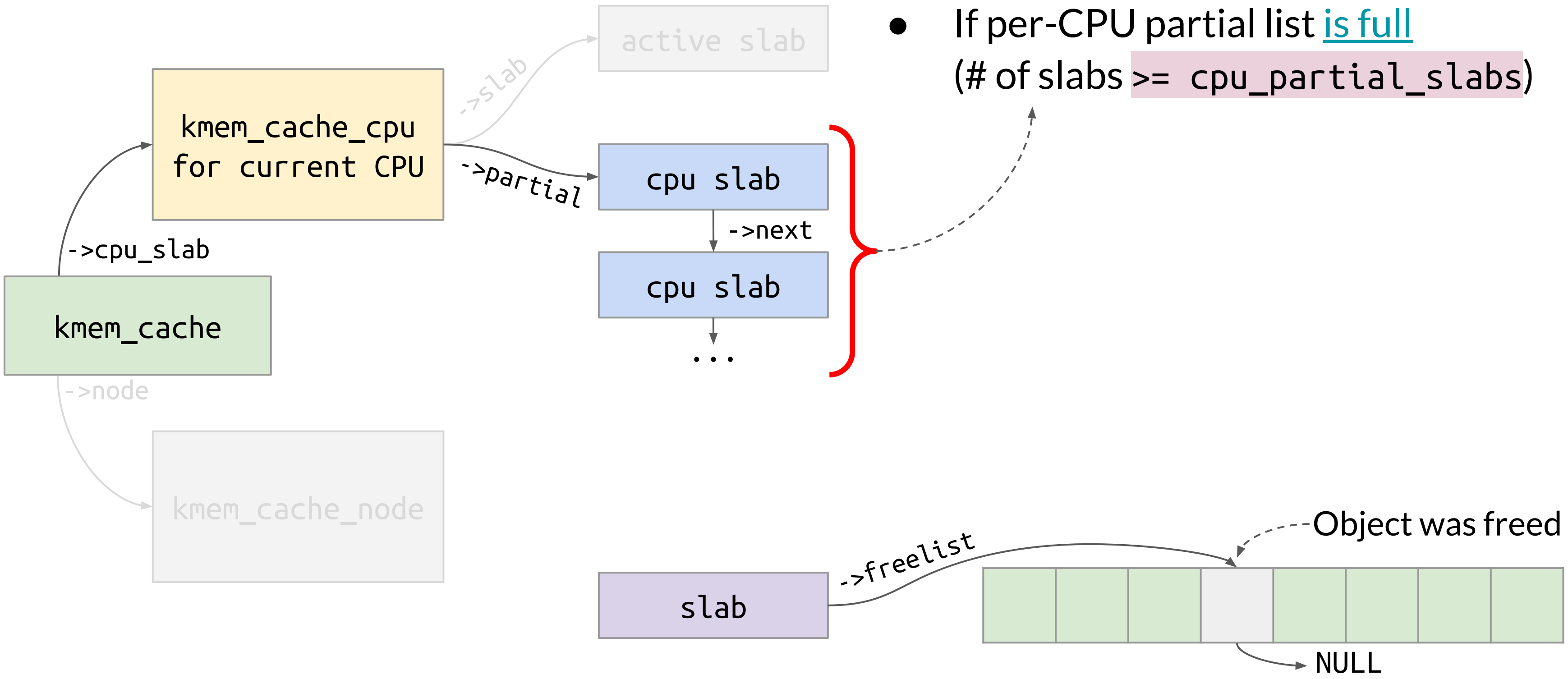
- If per-CPU partial list has space (# of slabs `< cpu_partial_slabs`)

Subcase #1: Per-CPU partial list does not overflow [1/1]



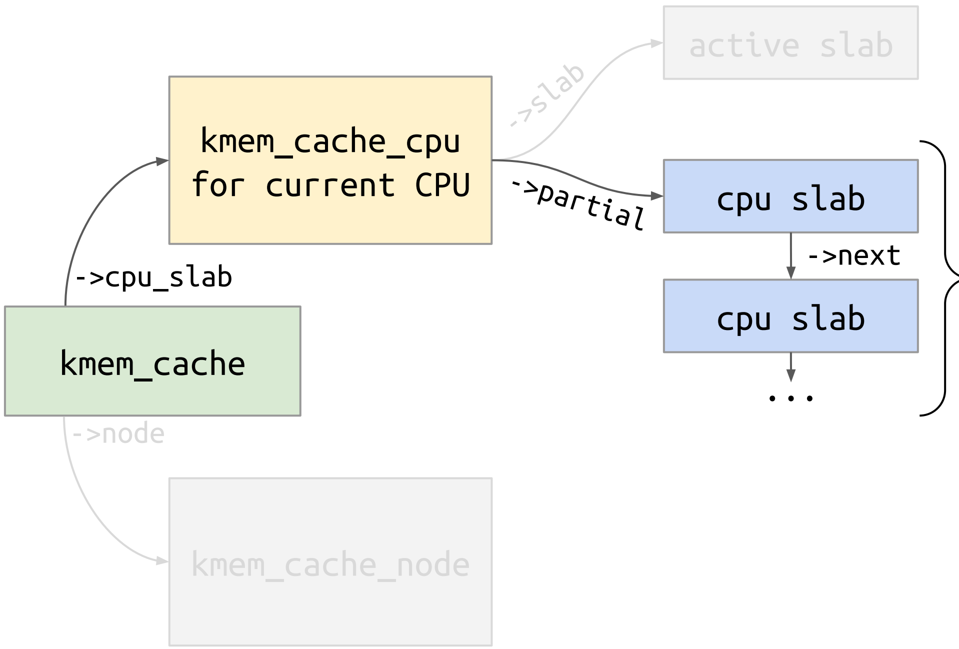
- If per-CPU partial list has space,
  - Put slab first onto per-CPU partial list for current CPU

Subcase #2: Per-CPU partial list overflows [0/4]



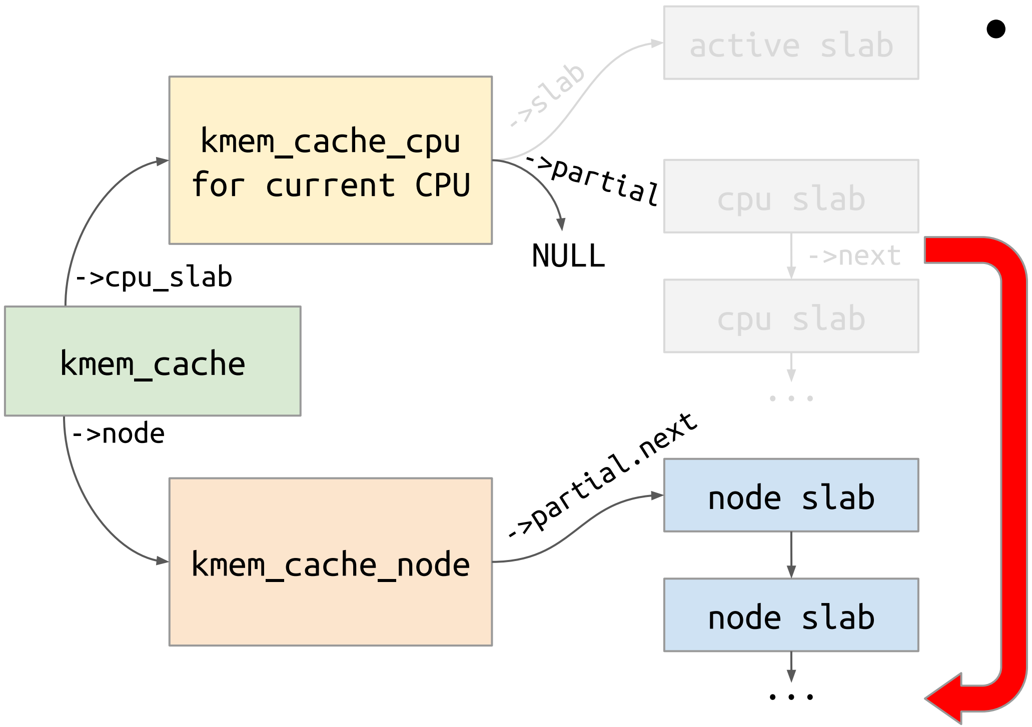
- If per-CPU partial list is full (# of slabs `>= cpu_partial_slabs`)

Subcase #2: Per-CPU partial list overflows [1/4]



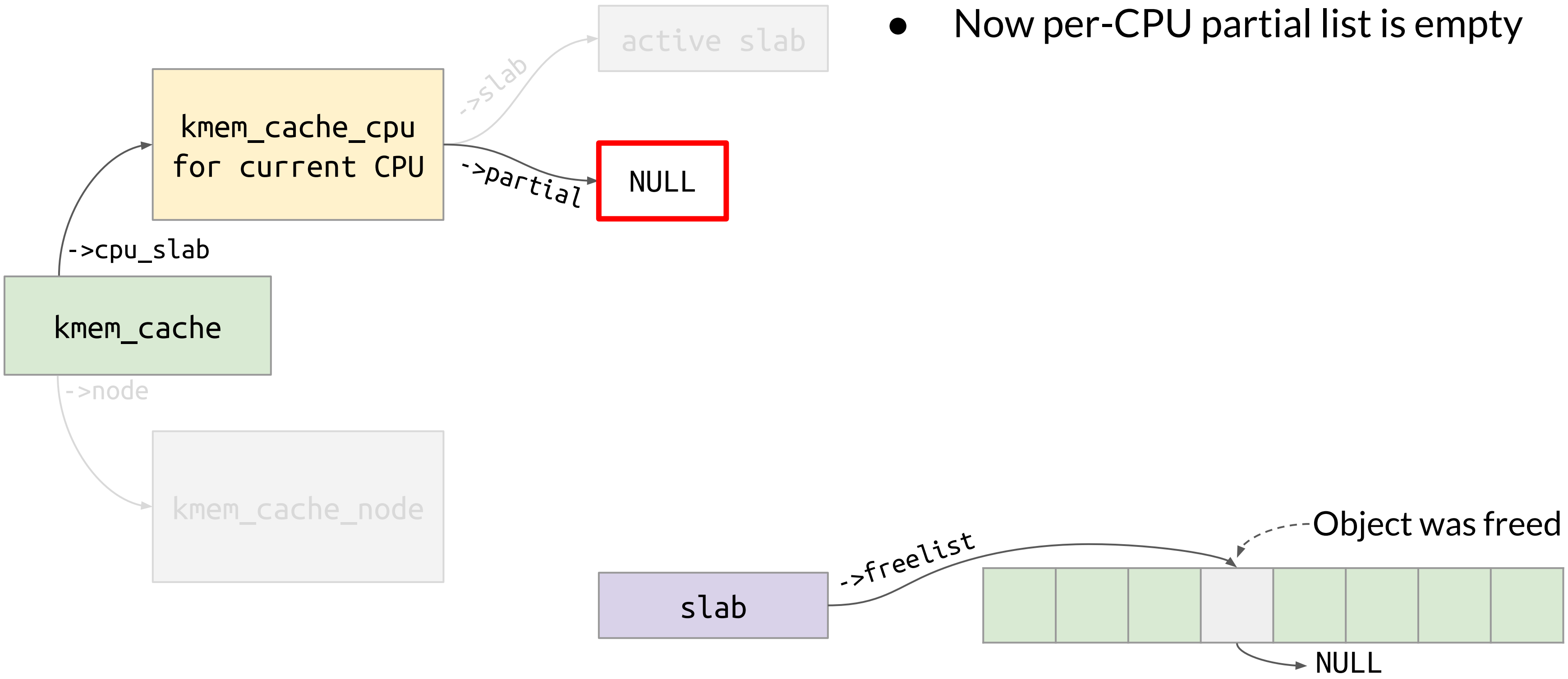
- If per-CPU partial list is full,
  free up per-CPU partial list
  (aka `unfreeze_partials` until 6.8)

Subcase #2: Per-CPU partial list overflows [2/4]



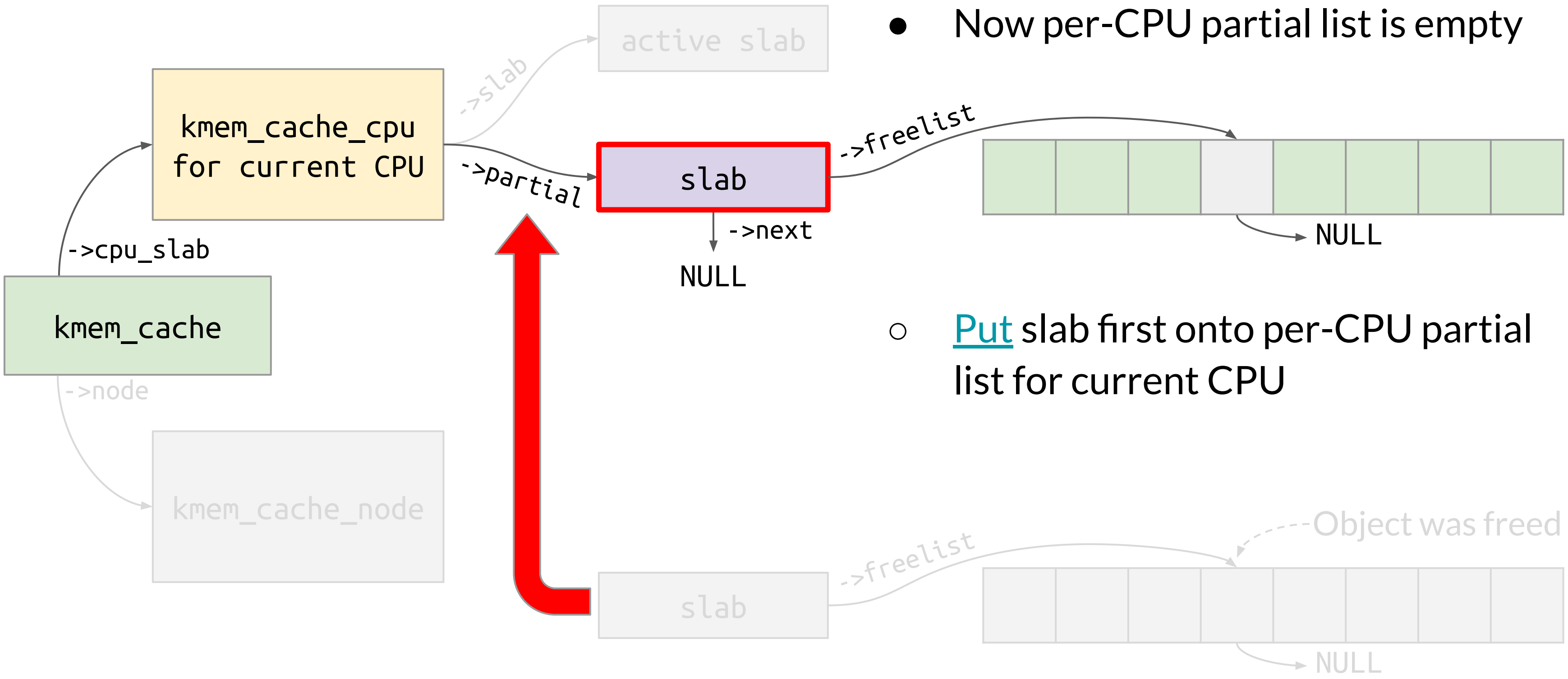
- If per-CPU partial list is full, free up per-CPU partial list:
  - Move per-CPU slabs to end of per-node list
  - Free empty per-CPU slabs to page_alloc once number of per-node slabs reaches `min_partial` (used for cross-cache)

Subcase #2: Per-CPU partial list overflows [3/4]



- Now per-CPU partial list is empty

Subcase #2: Per-CPU partial list overflows [4/4]



- Now per-CPU partial list is empty
  - Put slab first onto per-CPU partial list for current CPU

Exploitation-relevant outcomes

1. Freeing object in per-CPU partial slab puts that object first onto slab freelist
   - Slab might become empty, but it will stay on the list
     (used for cross-cache, out-of-scope)

2. Freeing object in full slab puts that slab first onto per-CPU partial list

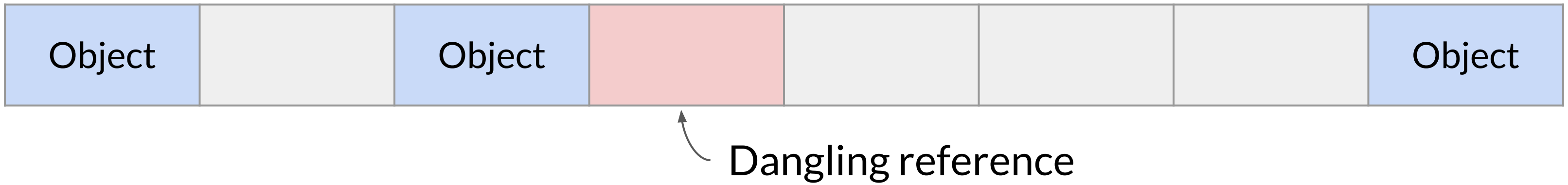
3. Overflowing per-CPU partial list frees some empty slabs on that list
   (used for cross-cache, out-of-scope)

Shaping Slab memory: Use-after-free, case #2

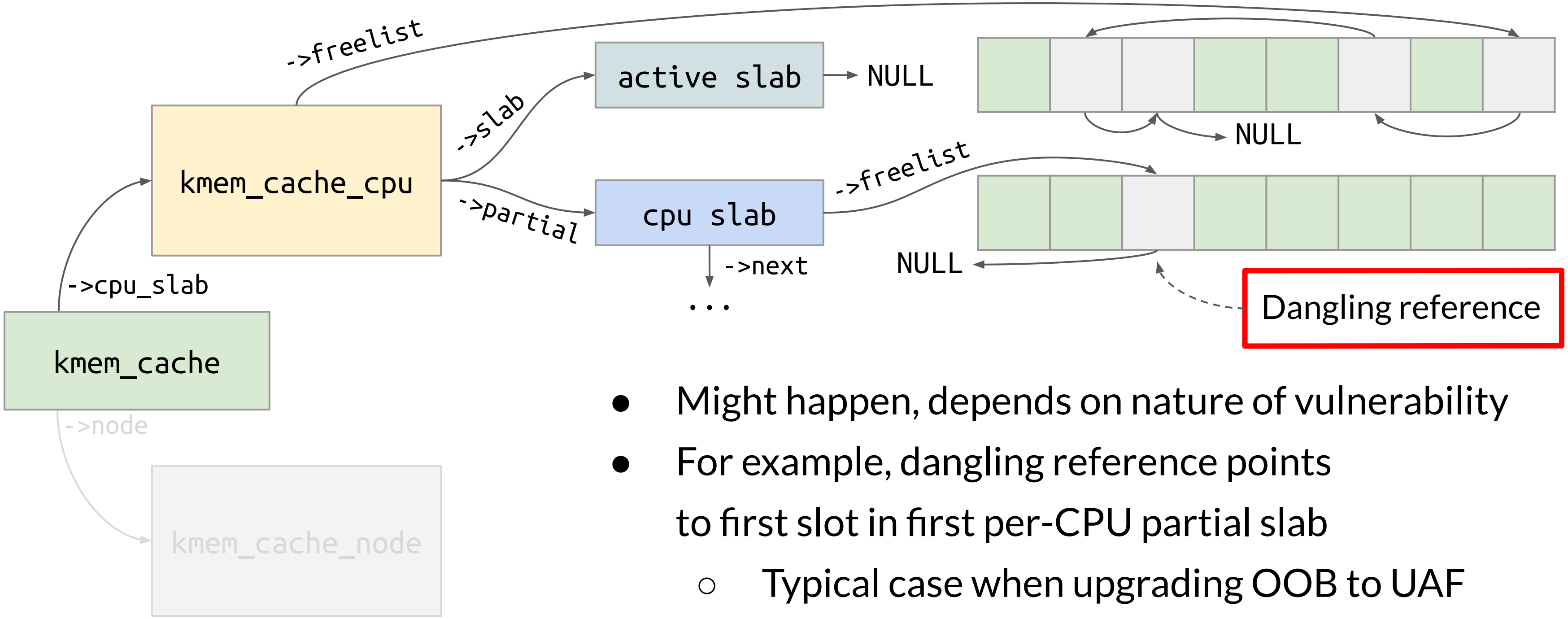
Reminder: Slab shaping for UAF on active slab

1. Allocate one vulnerable object via `IOCTL_ALLOC`
   - Object allocated from active slab
2. Free vulnerable object via `IOCTL_FREE`, dangling reference remains
   - Slot placed first onto lockless per-CPU freelist of active slab
3. ...

Active slab with vulnerable object freed:



What if UAF reference does not point to first in active?

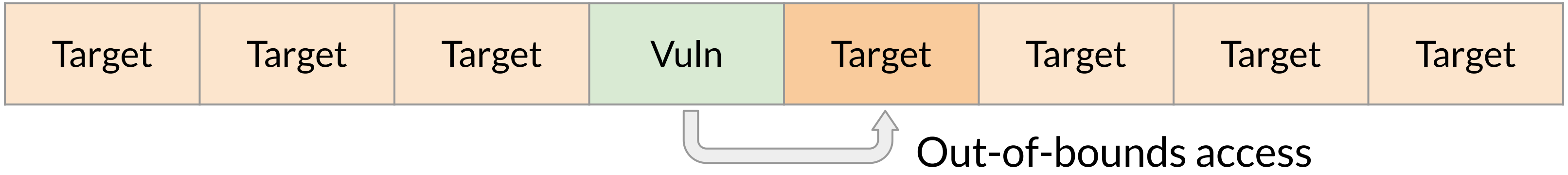


- Might happen, depends on nature of vulnerability
- For example, dangling reference points to first slot in first per-CPU partial slab
  - Typical case when upgrading OOB to UAF (see next slides)

Reminder: Slab shaping for simple OOB

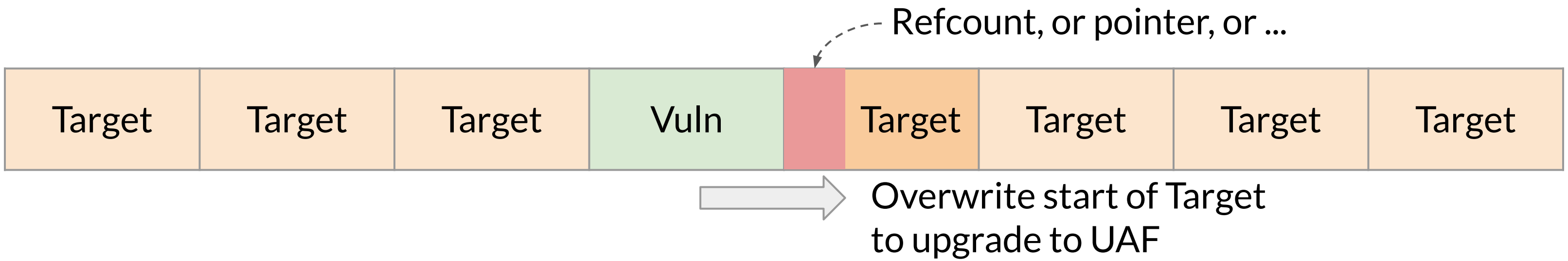
1. Allocate enough target objects to get new active slab
2. Allocate one vulnerable object via `IOCTL_ALLOC`
3. Allocate enough target objects to fill active slab
4. Trigger out-of-bounds access via `IOCTL_OOB`

Out-of-bounds from vulnerable object lands in target object:



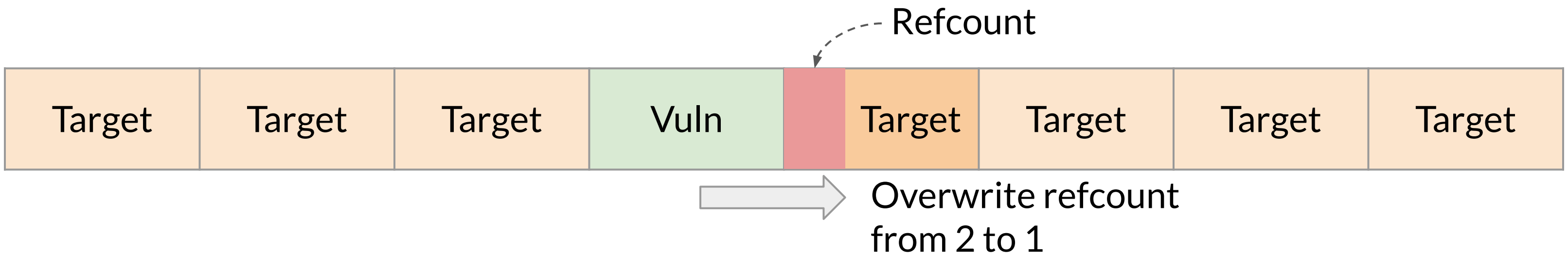
Upgrading OOB to UAF

- Out-of-bounds write is relatively weak primitive
  - Might want to upgrade to use-after-free
  - By overwriting reference counter, pointer, or something else in target object



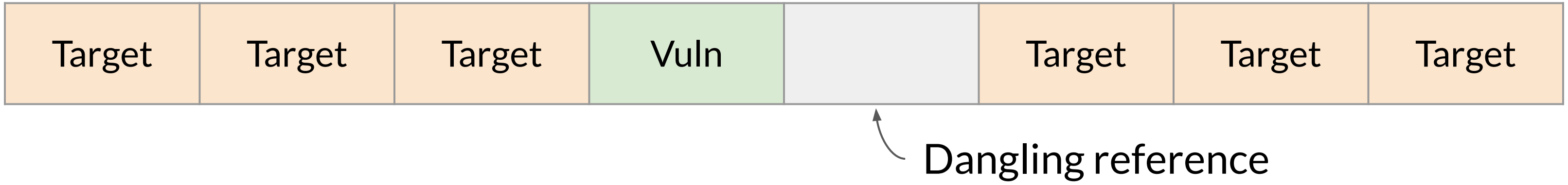
Example of upgrading OOB to UAF [1/2]

1. Overwrite reference counter in target object from 2 to 1 (e.g.)
    - Via previously discussed Slab shaping technique for simple OOB
    - Assume each target object has initial refcount of 2
    - Slab becomes full after shaping



Example of upgrading OOB to UAF [2/2]

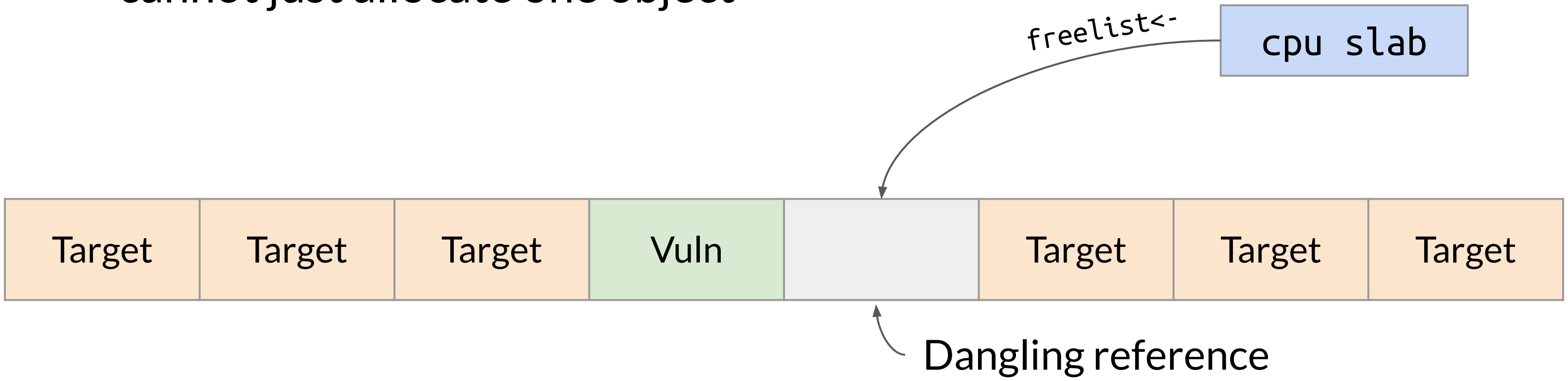
1. Overwrite reference counter in target object from 2 to 1 (e.g.)
2. Decrement refcount once for each allocated target object
   - Assume we have userspace reference to each, like file descriptor
   - Object with overwritten refcount gets freed
   - Now have dangling reference to slot



How to reclaim UAF slot after upgrading?

- Slab was full before last free ⇒ Slab goes to per-CPU partial list

- How to reclaim freed slot in slab on per-CPU partial list?
  - Slot is not last on lockless per-CPU freelist,
    cannot just allocate one object



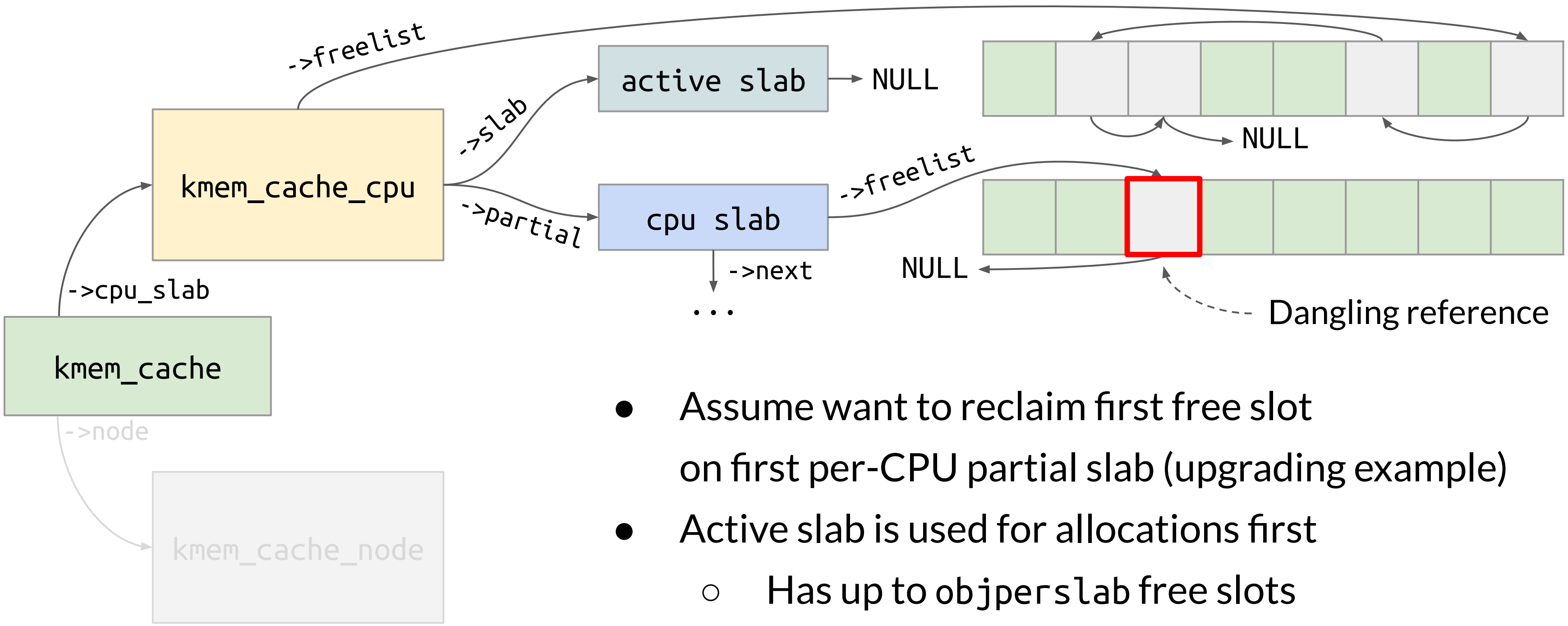
Solution: Slab spraying

- Spraying is Slab shaping technique
  - Allocating many objects with goal that one will be placed in target slot

- Similar to "allocating many objects to plug holes"
  - I prefer term "spraying" when there is a target slot

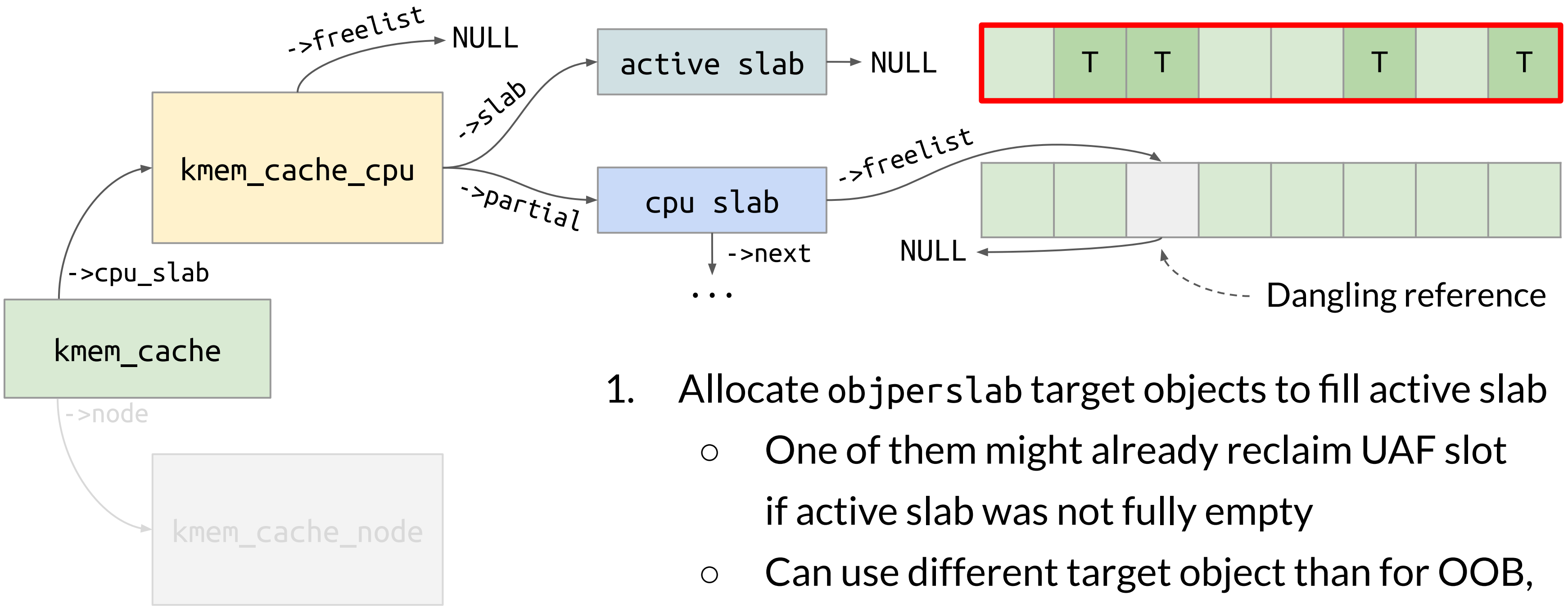
- How many objects to spray?
  - Depends on nature of dangling reference:
    to which slot of which partial slab it points

Reclaiming first slot on first per-CPU partial slab [0/2]



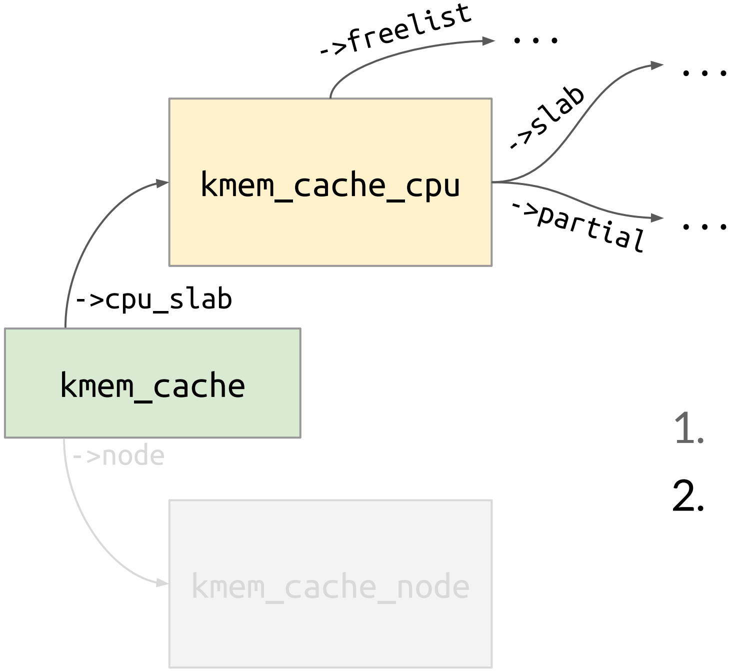
- Assume want to reclaim first free slot on first per-CPU partial slab (upgrading example)
- Active slab is used for allocations first
  - Has up to `objperslab` free slots

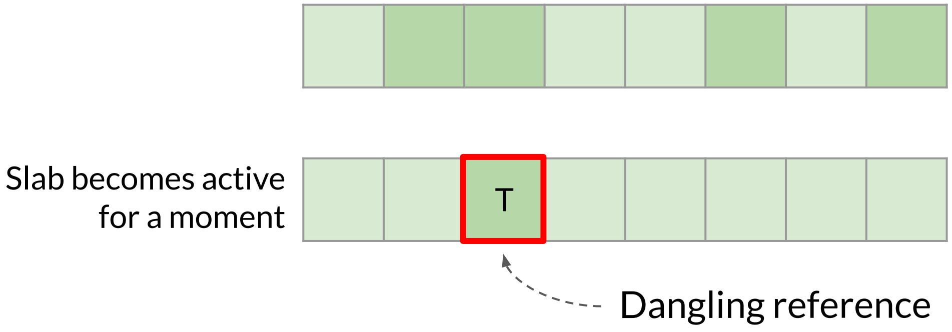
Reclaiming first slot on first per-CPU partial slab [1/2]



1. Allocate `objperslab` target objects to fill active slab
    - One of them might already reclaim UAF slot if active slab was not fully empty
    - Can use different target object than for OOB, one that is useful for UAF exploitation

Reclaiming first slot on first per-CPU partial slab [2/2]





1. Allocate `objperslab` target objects to fill active slab
2. Allocate one more target object
   - Object reclaims UAF slot
     if original active slab was full empty
   - Now can trigger UAF

Shaping Slab memory: Out-of-bounds, case #2 again

Case #2: Combined allocation and out-of-bounds trigger

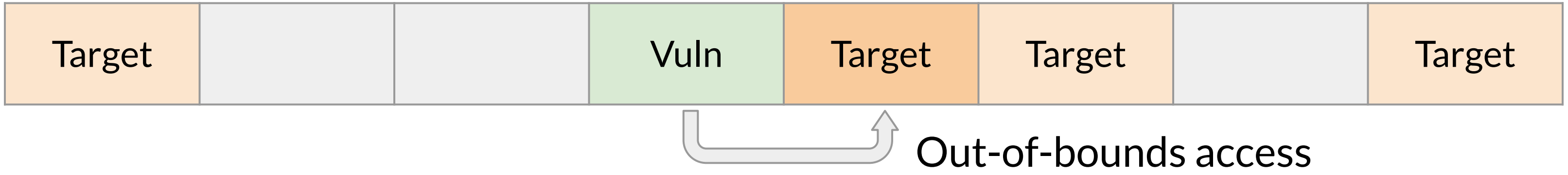
- Case #1:
  - `IOCTL_ALLOC` — Allocates vulnerable object
  - `IOCTL_OOB` — Writes or reads data out-of-bounds of vulnerable object

- Case #2:
  - `IOCTL_ALLOC_AND_OOB` — Allocates vulnerable object and immediately writes data out-of-bounds

Reminder: Allocation-only approach for combined OOB

1. Allocate enough target objects to get new active slab
2. Allocate one vulnerable object and trigger OOB via `IOCTL_ALLOC_AND_OOB`

- If out-of-bounds access lands in target object
  - Success: target object overwritten
  - But might need multiple retries to achieve this



Making-holes approach for combined OOB [1/4]

1. Allocate enough target objects to get new active slab

Making-holes approach for combined OOB [2/4]

1. Allocate enough target objects to get new active slab
2. Allocate `objperslab` more target objects to fill slab
    - Slab becomes full

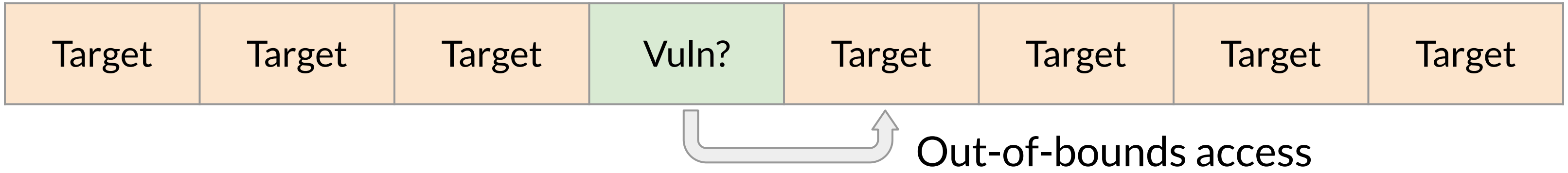
Target	Target	Target	Target	Target	Target	Target	Target

Making-holes approach for combined OOB [3/4]

1. Allocate enough target objects to get new active slab
2. Allocate `objperslab` more target objects to fill slab
3. Free one target object from this slab (make a hole)
   - Assume we have userspace reference to each, like file descriptor
   - Free target object that was allocated `objperslab` allocations ago

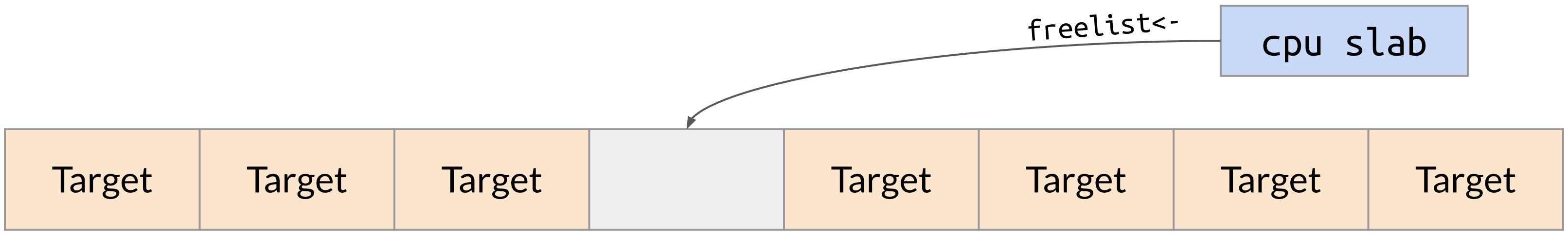
Making-holes approach for combined OOB [4/4]

1. Allocate enough target objects to get new active slab
2. Allocate `objperslab` more target objects to fill slab
3. Free one target object from this slab (make a hole)
4. Reclaim free slot with vulnerable and trigger OOB via `IOCTL_ALLOC_AND_OOB`?
    - Tricky!



Trickiness with step 4 [1/2]

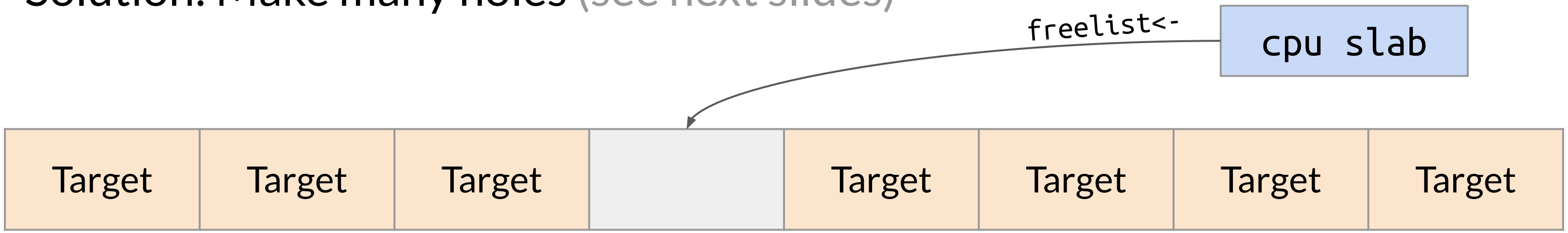
- After step 3, freed slot belongs to per-CPU partial slab
- Have to plug holes in new active slab, allocations come from there first



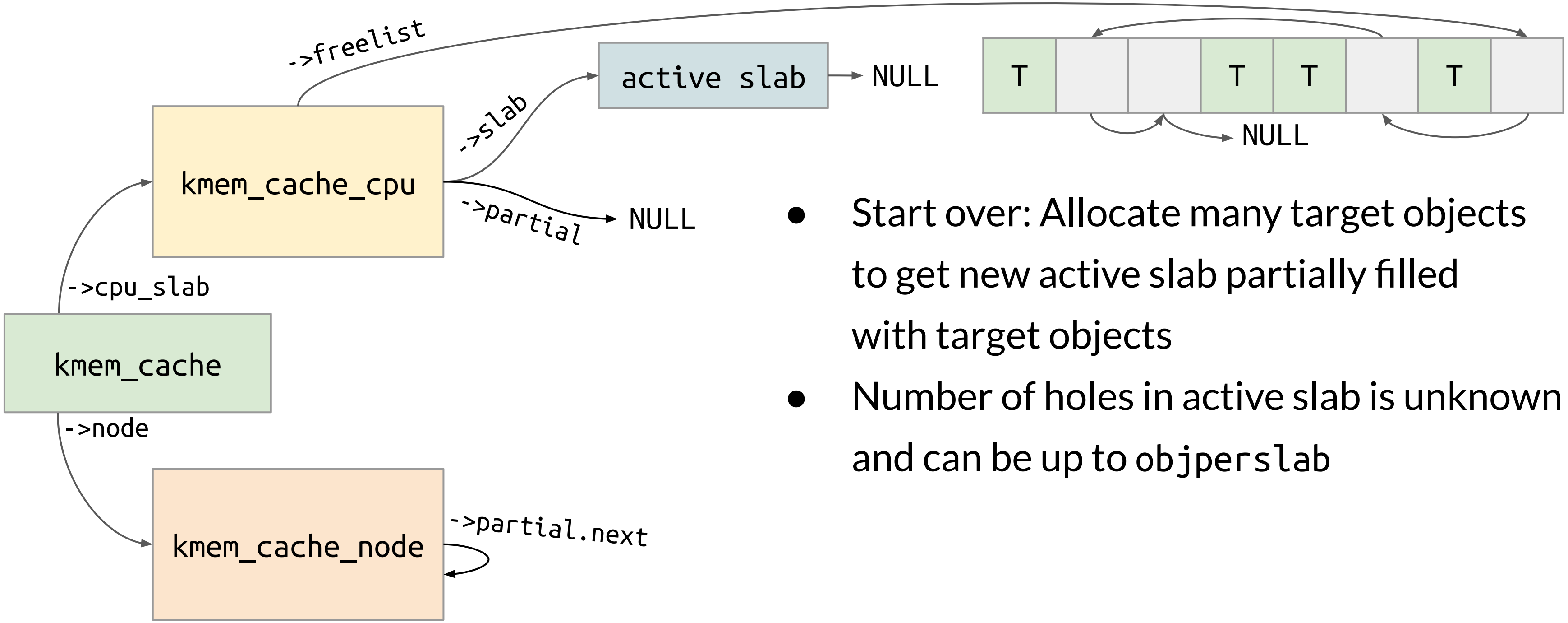
Trickiness with step 4 [2/2]

- Cannot allocate multiple vulnerable objects to plug holes in active slab
  - Might be impossible due to nature of vulnerability
  - Also each vulnerable object allocation triggers OOB: defeats purpose of making-hole approach

- Solution: Make many holes (see next slides)

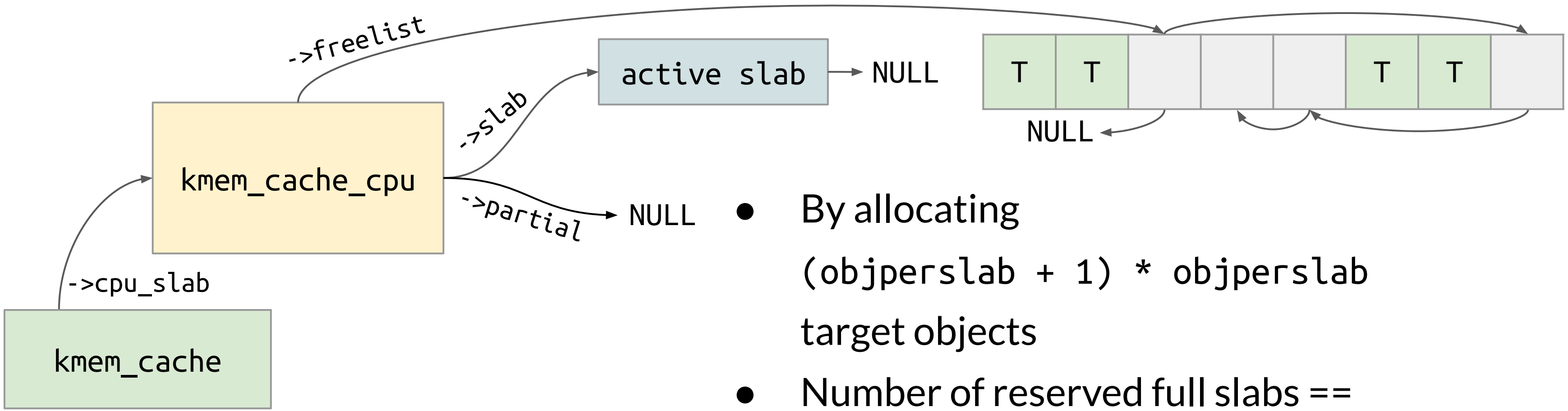


1: Plug all holes and get new active slab with target objects

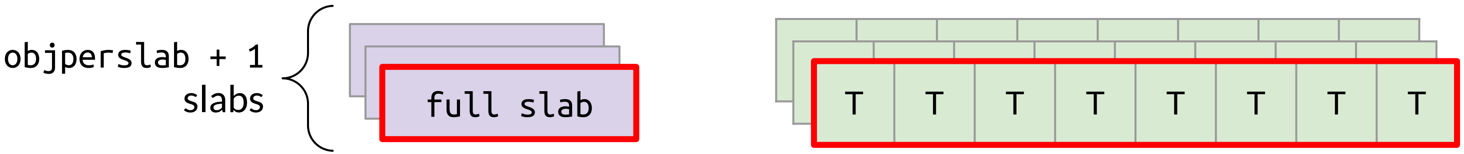


- Start over: Allocate many target objects to get new active slab partially filled with target objects
- Number of holes in active slab is unknown and can be up to `objperslab`

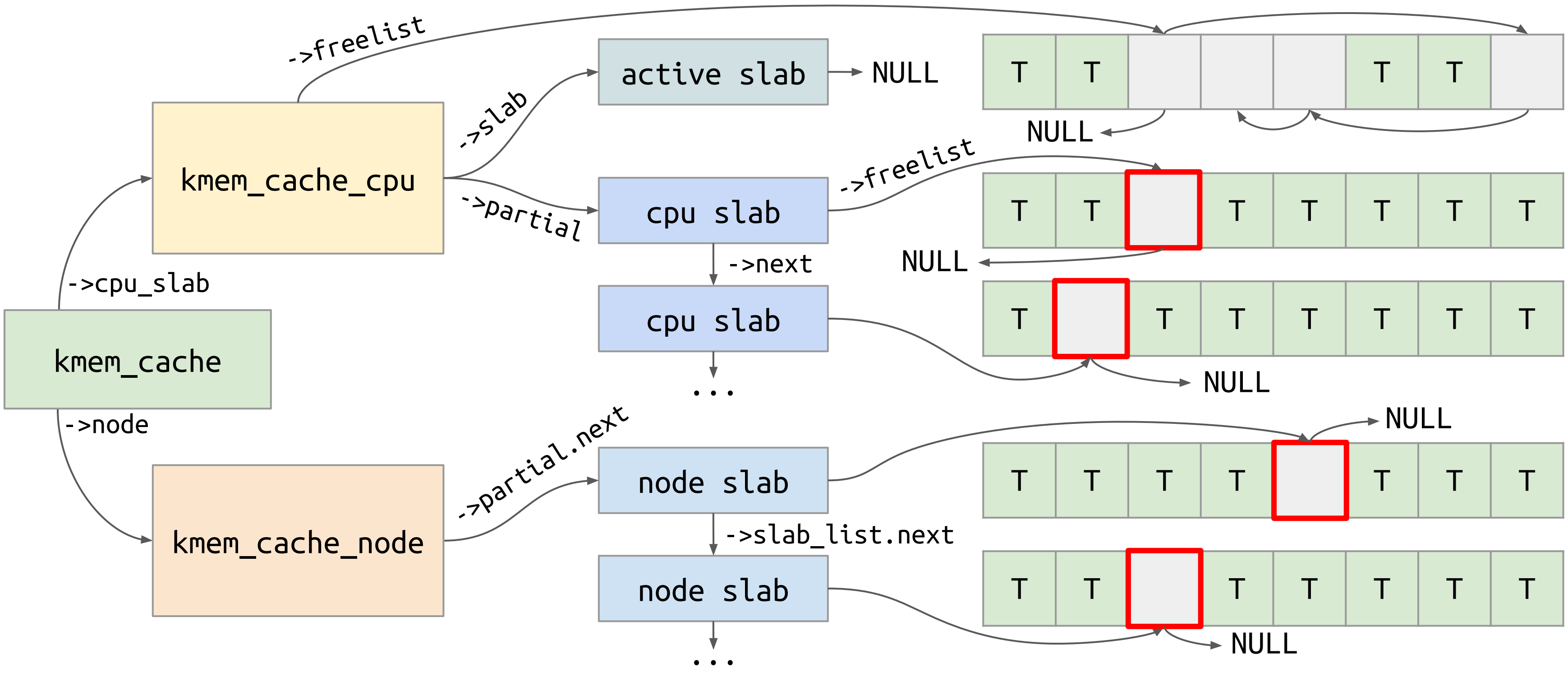
2: Reserve `objperslab + 1` full slabs with target objects



- By allocating `(objperslab + 1) * objperslab` target objects
- Number of reserved full slabs == potentially possible number of free slots in active slab + 1

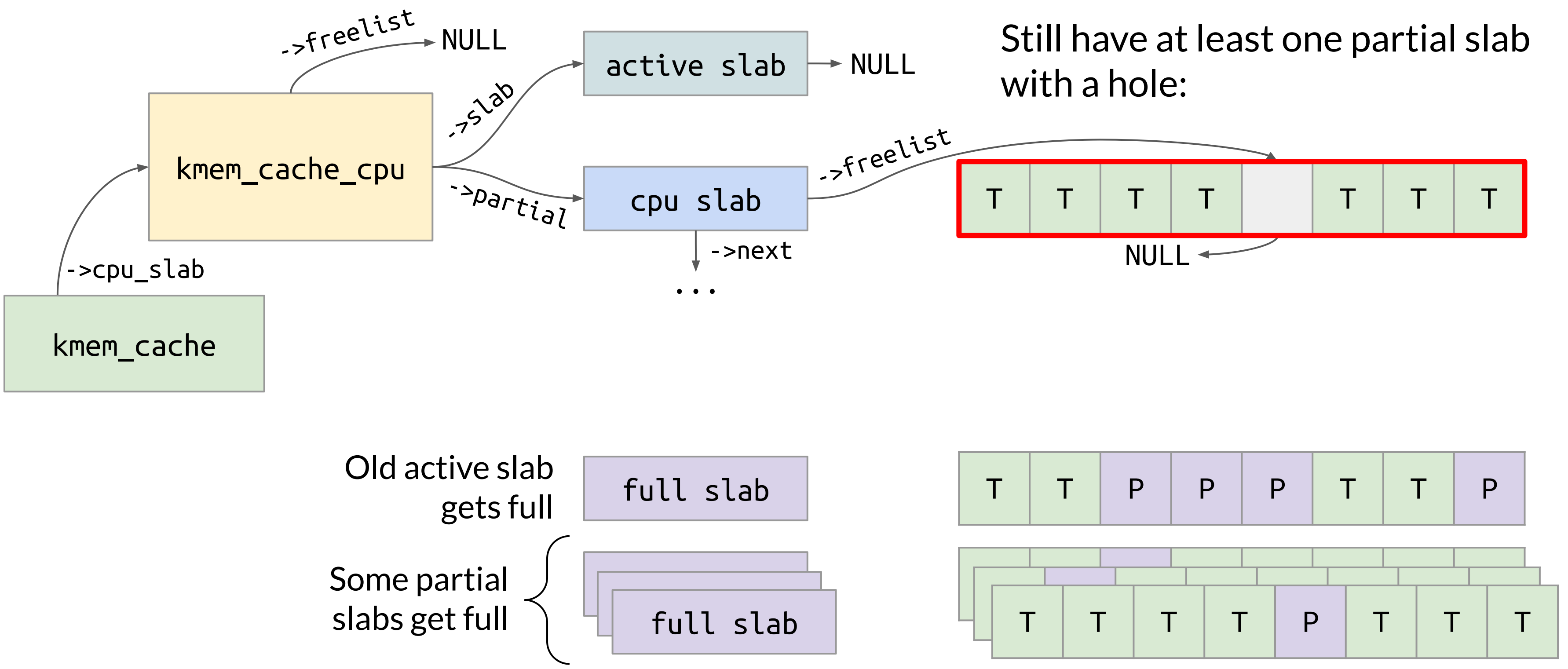


3: Make one hole in each reserved full slab

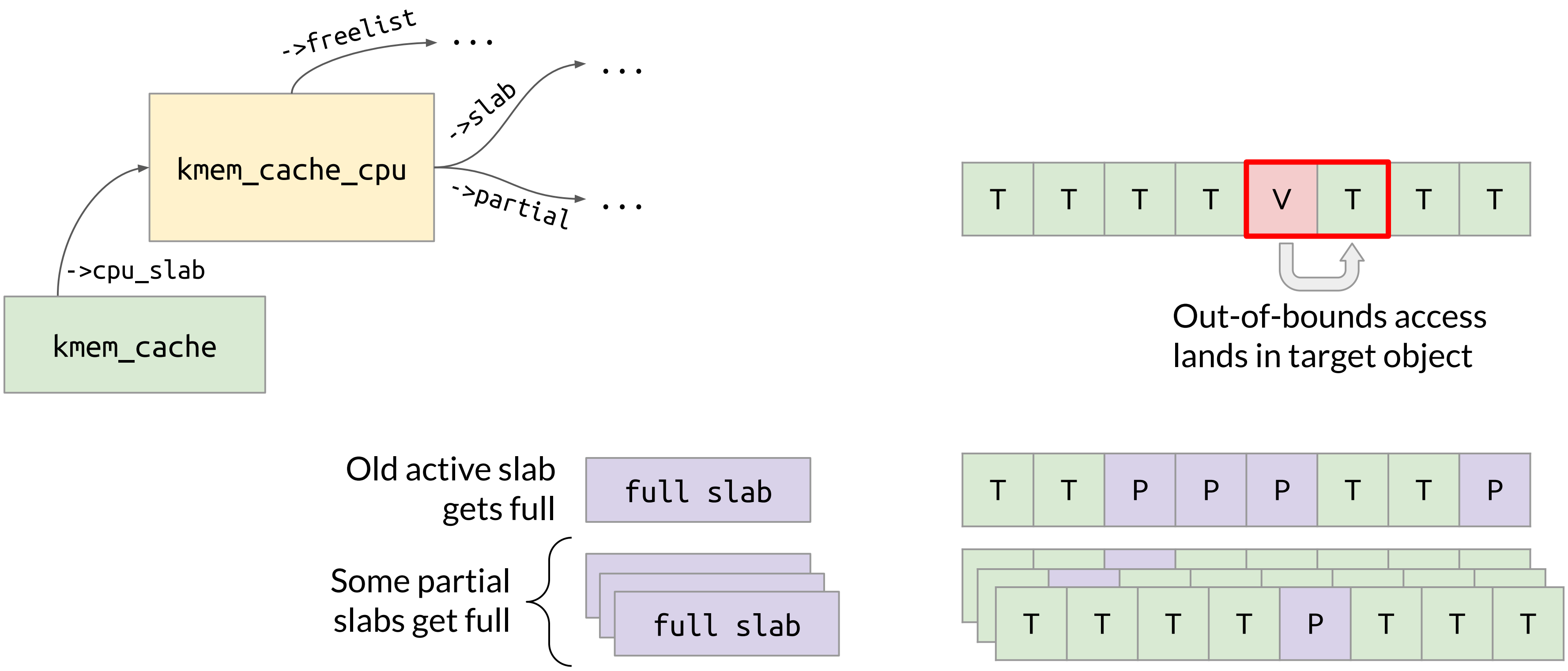


`objperslab + 1 > cpu_partial_slabs` for any cache ⇒ Some slabs will end up on per-node list

4: Allocate `objperslab` plug objects (target objects work too)



5: Allocate one vulnerable object and trigger OOB



Making-holes vs allocation-only for combined OOB

- Upside of making-holes approach:
  - Works for OOB that could corrupt freelist pointer in free slot

- Downsides of making-holes approach:
  - Allocating many full slabs takes time
    ⇒ Higher chance of failures due to preemption
  - Interacts with per-node partial list
    ⇒ Less reliability as other CPUs interact with this list too

Double-free

Exploiting double-frees

- Typical approach: convert to use-after-free:
    1. Free vulnerable object once
    2. Replace freed slot with transient object
    3. Free vulnerable object again via double-free
    4. Now have UAF reference via transient object

- Immediate double-free is detected with `CONFIG_SLAB_FREELIST_HARDENED=y`
- Freeing free slot via double-free can be used for cross-cache attacks

Final notes

SLUB is complicated

- Details shown on these slides is a model
  - Good enough as a start
  - But SLUB has many special cases and optimizations

- Read SLUB code to learn specifics
  - `__slab_alloc_node` — Starting point for allocation process
  - `do_slab_free` — Starting point for freeing process

Further reading: SLUB and cross-cache

- More details about how SLUB works:
  - Linux SLUB Allocator Internals and Debugging [article] [note]
- About cache merging, accounting, and hardened usercopy:
  - Linux kernel heap feng shui in 2022 [article]
- Introduction to cross-cache use-after-free attacks:
  - CVE-2022-29582, An io_uring vulnerability [article]

Further reading: Improving reliability of Slab shaping

- Playing for K(H)eaps: Understanding and Improving Linux Kernel Exploit Reliability [paper]

- PSPRAY: Timing Side-Channel based Linux Kernel Heap Exploitation Technique [paper]

- SLUBStick: Arbitrary Memory Writes through Practical Software Cross-Cache Attacks within the Linux Kernel [paper]

💜 Thank you!