black hat ASIA 2024

APRIL 18-19, 2024

BRIEFINGS

# Game of Cross Cache: Let's win it in a more effective way!

Le Wu From Baidu Security

# About me

- Le Wu, @NVamous on Twitter
- Focus on Android/Linux vulnerability
- Dirty Pagetable —— A novel technique to rule the Linux Kernel [1]
- Blackhat USA, Europe, Asia

[1]:https://yanglingxi1993.github.io/dirty_pagetable/dirty_pagetable.html

# Agenda

- Introduction to Cross-cache attack
- Challenges in Cross-cache attack
- Advancing Towards a More Effective Cross-cache Attack
- Exploit File UAF with Dirty Pagetable
- Summary

# Introduction to Cross-cache attack

## A Simplified Cross-cache Attack For UAF

**UAF**

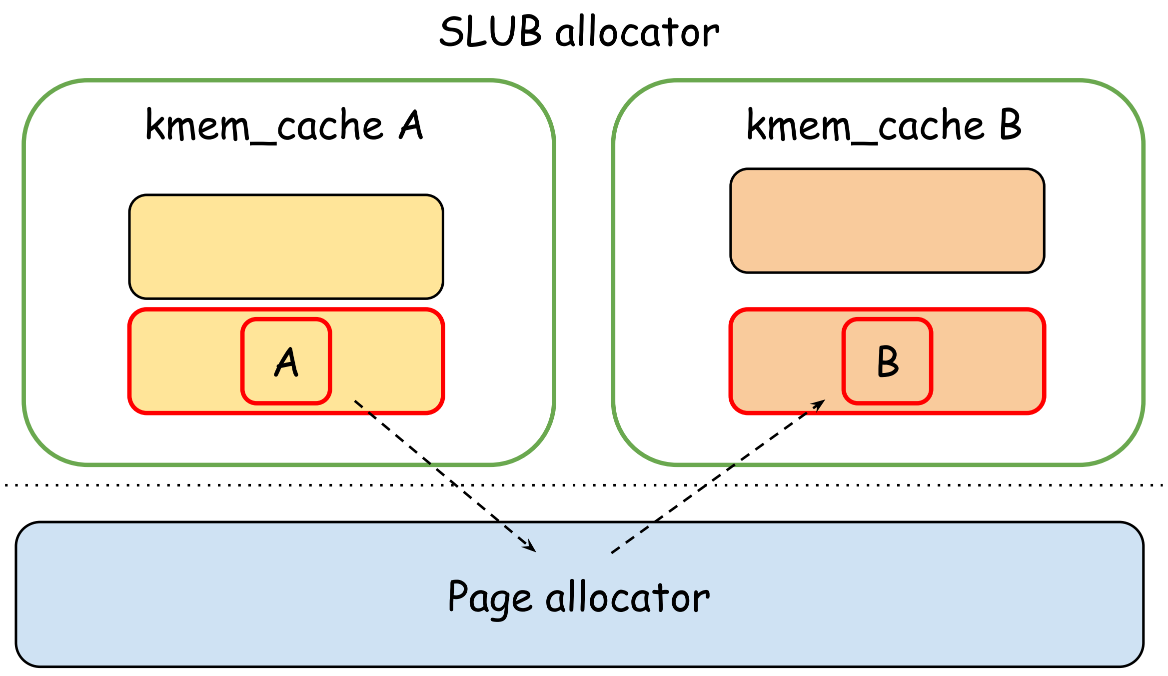
Trigger UAF to release the victim **object A**;

Reclaim the victim slab of victim **object A** to Page allocator;

kmem_cache B reuse the pages of victim slab, and **object A** is reallocated as **object B**;

Operations to victim **object A**; → corrupt the **object B**

Make use of corrupted **object B** to get ROOT;




(Object A or object B could be pages or other kinds of memory regions)

# Introduction to Cross-cache attack

Cross-cache attack is getting popular:

- Original vulnerable object is not exploitable, especially the one allocated from a dedicated kmem_cache
- Transform the unknown vulnerability to well-known one to simplify the exploitation
- Build data-only exploitation techniques to defeat growing mitigations like KASLR, PAN, CFI...

| Method | Cross-cache From | Cross-cache To |
| --- | --- | --- |
| ret2dir | * | direct mapping |
| ret2page | * | kernel allocated page |
| Drity Cred | * | struct cred |
| **Dirty Pagetable** | * | **user page table** |
| ... | ... | ... |

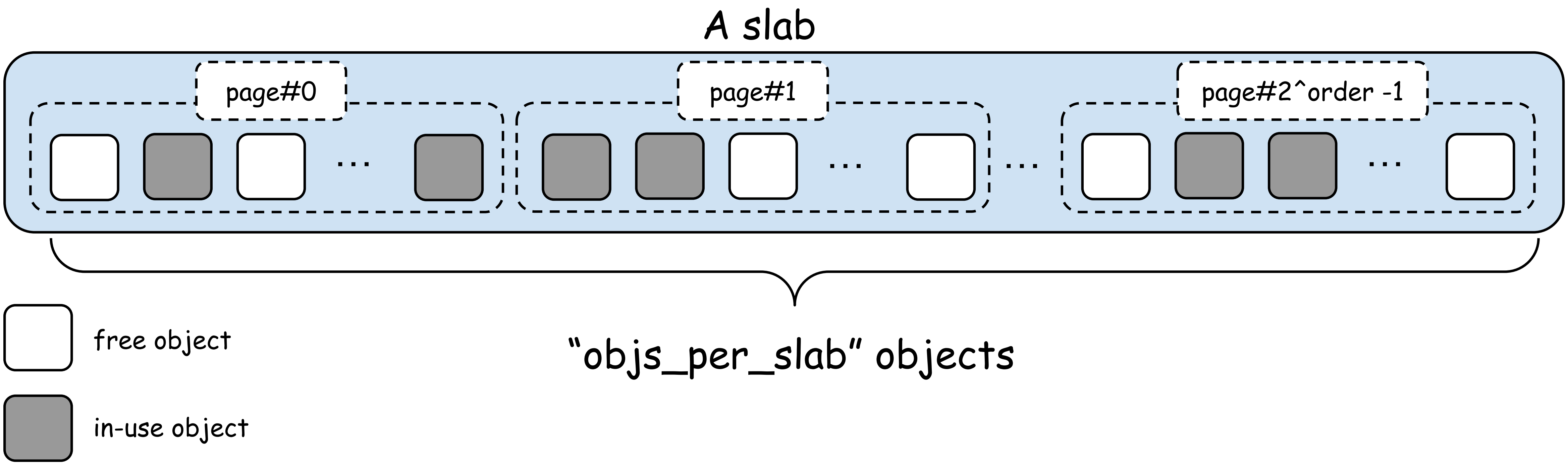
# Introduction to Cross-cache attack

Can we make it less unstable, or in other words, more efficient?

Well, it's known as an **unstable** technique...

# Common workflow of Cross-cache attack

Step0. Common knowledge for SLUB allocator

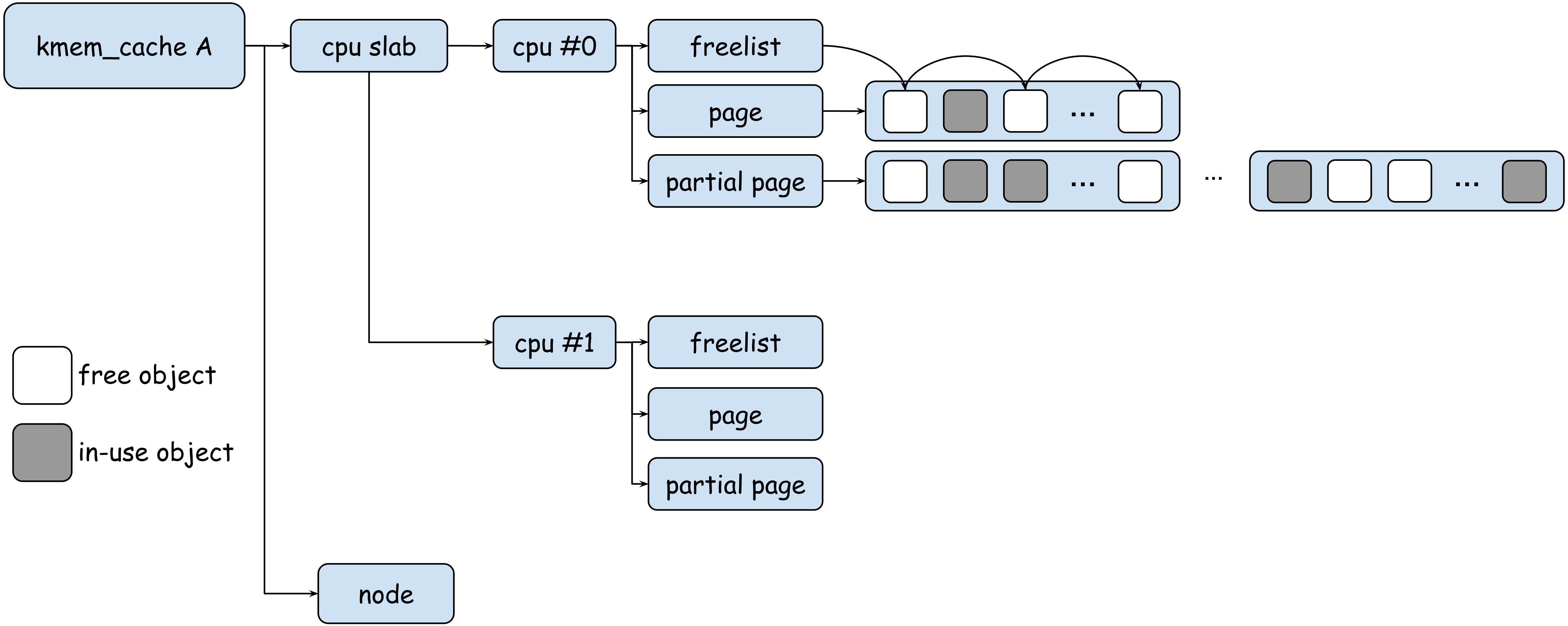



**objs_per_slab**: number of objects in a single slab
**order**: order of pages in a single slab

```
x1q:/sys/kernel/slab/kmalloc-256 # cat objs_per_slab
32
x1q:/sys/kernel/slab/kmalloc-256 # cat order
1
```

# Common workflow of Cross-cache attack

Step 0. Common knowledge for SLUB allocator

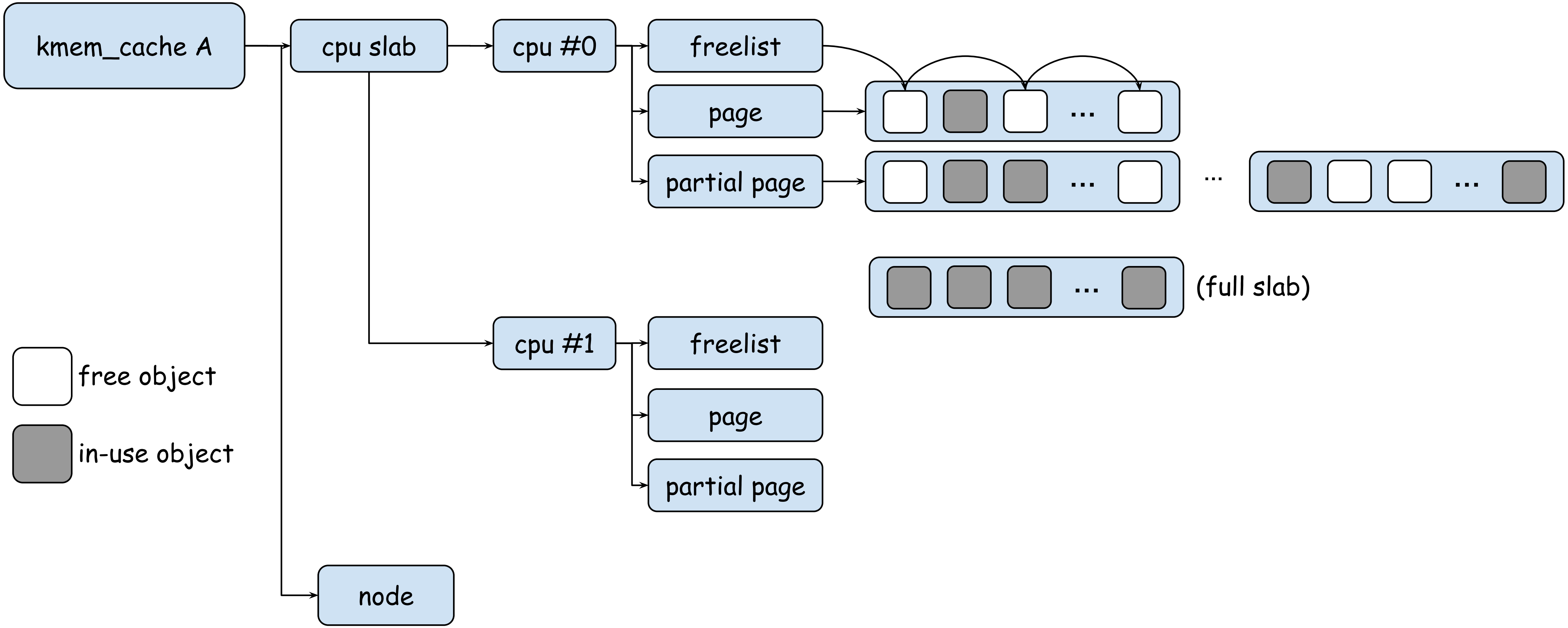


# Common workflow of Cross-cache attack

Step0. Common knowledge for SLUB allocator

The deterministic method for putting slab into the percpu partial list:

- Create a full slab

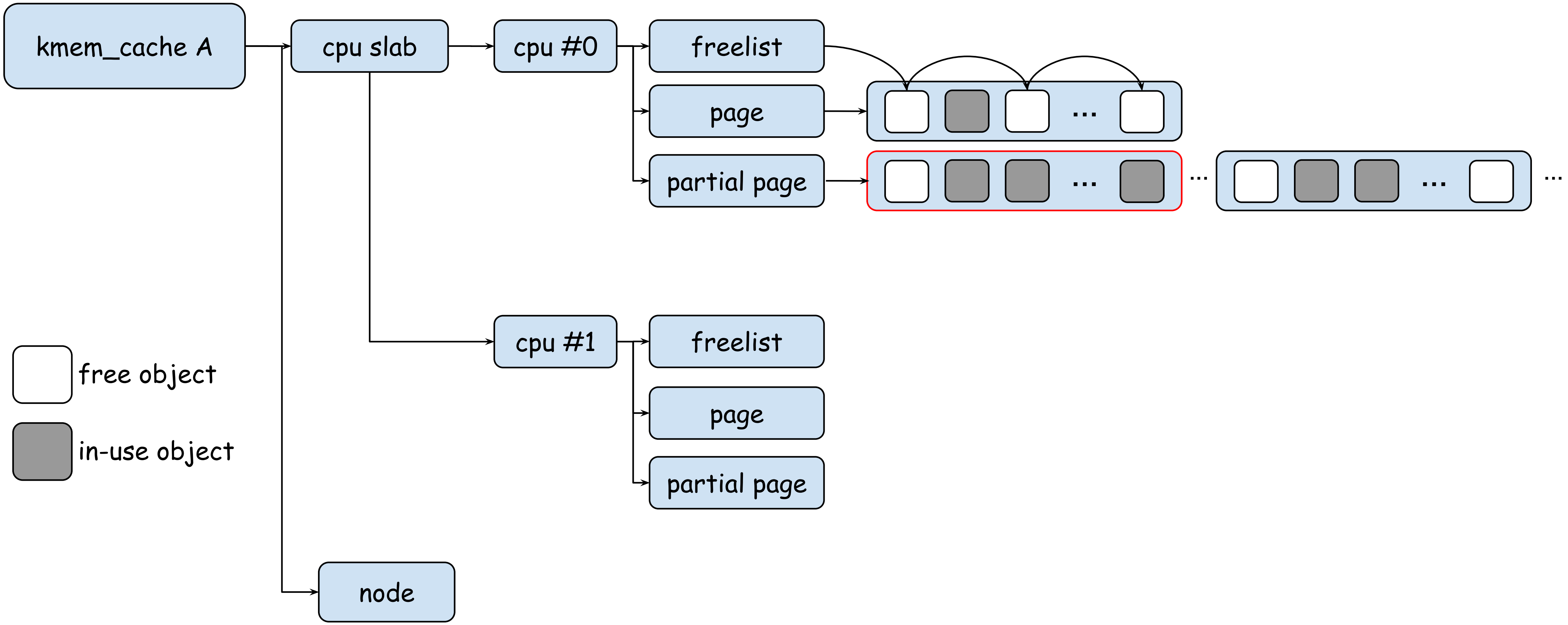


# Common workflow of Cross-cache attack

Step0. Common knowledge for SLUB allocator

The deterministic method for putting slab into the percpu partial list:

- Pin on cpu#0 and release an object from the full slab



# Common workflow of Cross-cache attack

Step0. Common knowledge for SLUB allocator

Flushing for the percpu partial list:

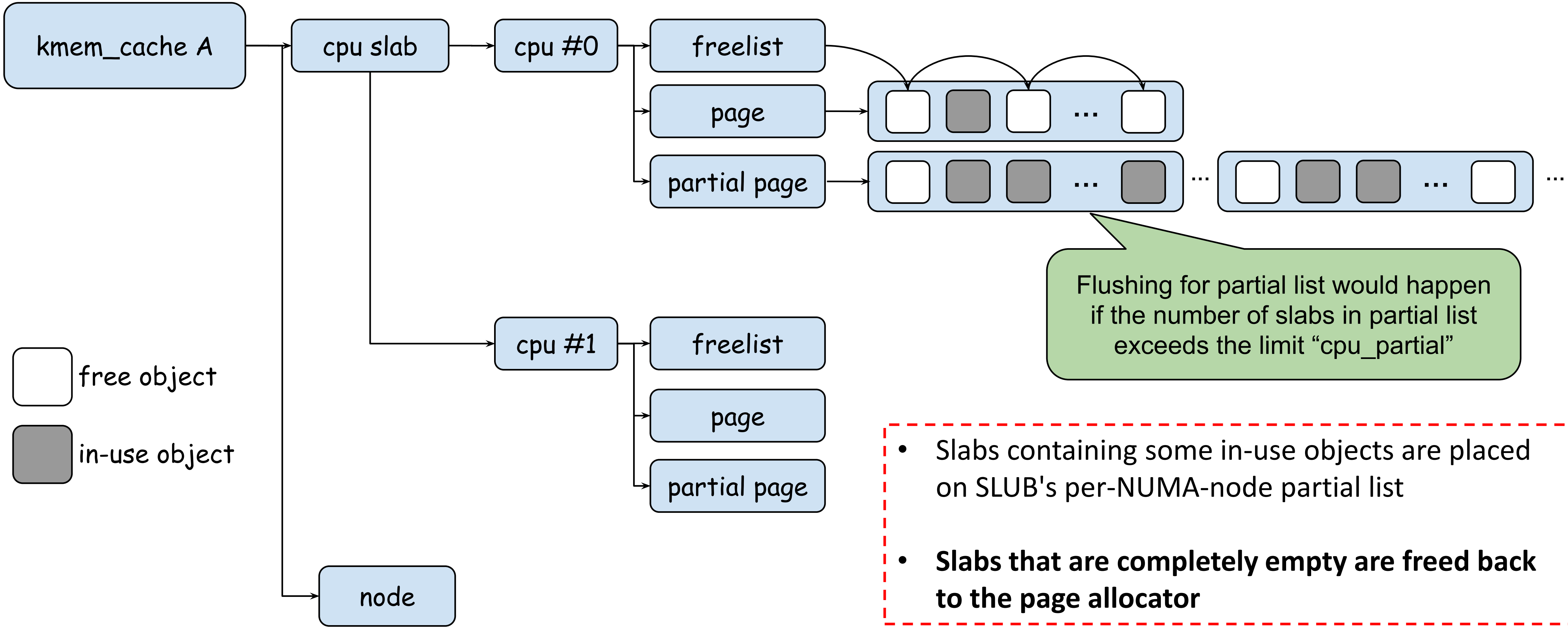
**cpu_partial:** the maximum number of  slabs can be put in the percpu partial list

```
x1q:/sys/kernel/slab/kmalloc-256 # cat cpu_partial
13
```

# Common workflow of Cross-cache attack

Step0. Common knowledge for SLUB allocator

Flushing for the percpu partial list:



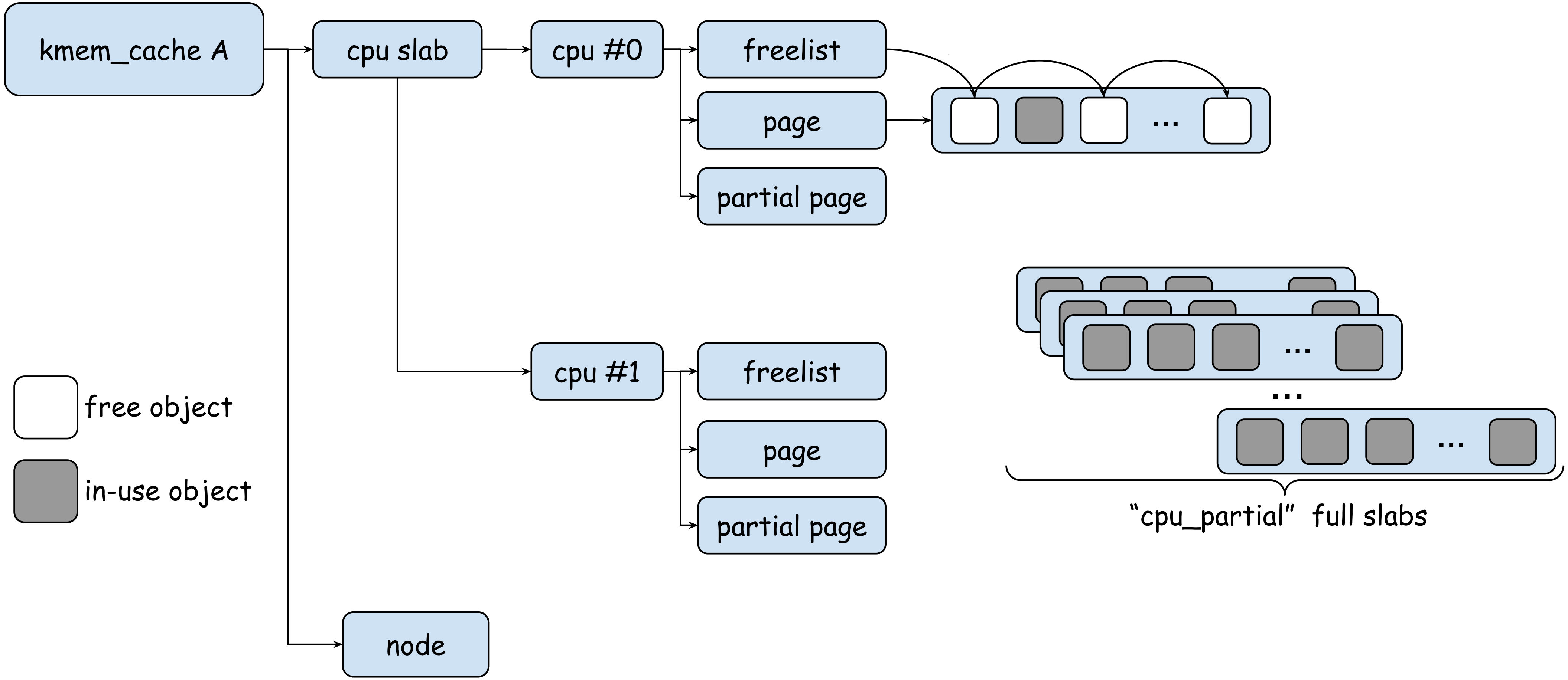

- Slabs containing some in-use objects are placed on SLUB's per-NUMA-node partial list
- **Slabs that are completely empty are freed back to the page allocator**

# Common workflow of Cross-cache attack [2]

Step1. Pin our task to a single CPU, for example, cpu#0

Step2. Defragmentation: to drain partially-free slabs of all their free objects

Step3. Allocate around **objs_per_slab** * (1+**cpu_partial**) objects



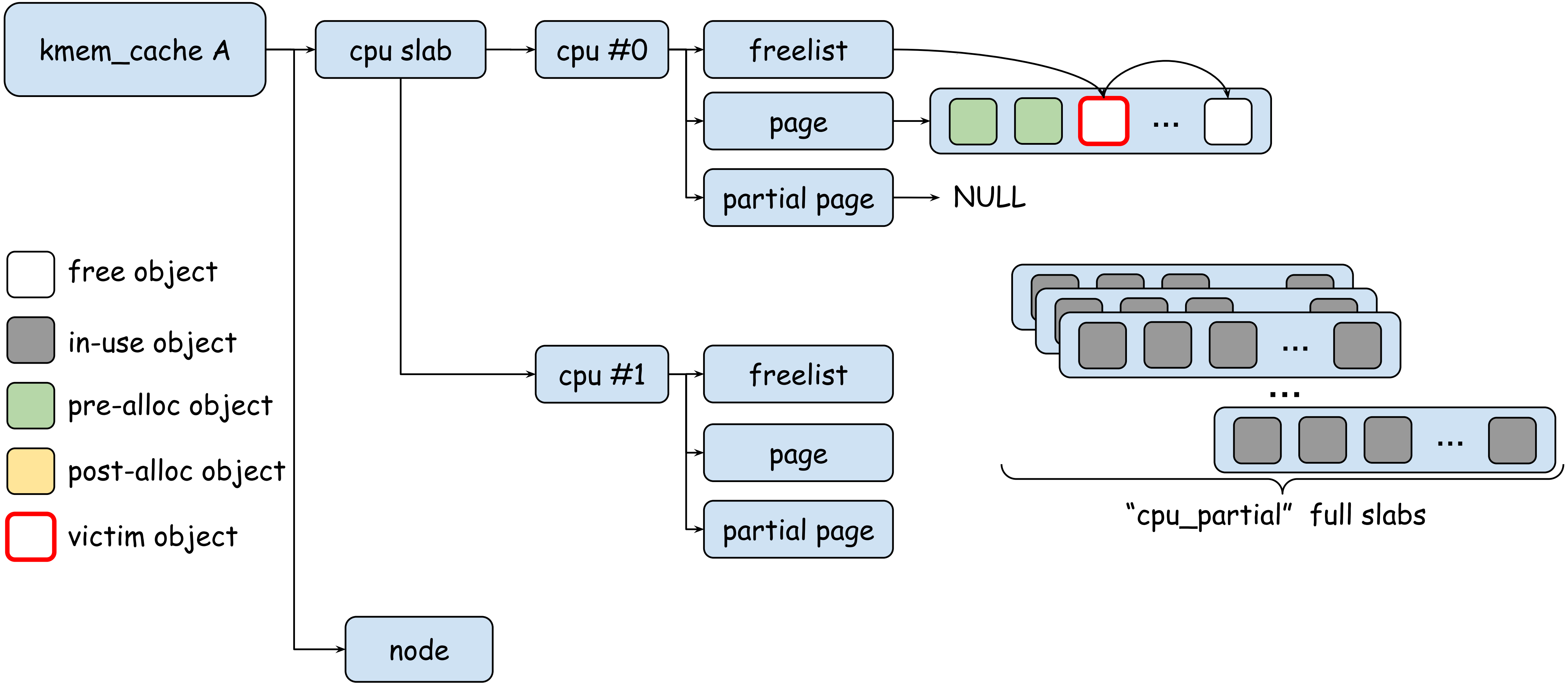

[2]:https://googleprojectzero.blogspot.com/2021/10/how-simple-linux-kernel-memory.html

# Common workflow of Cross-cache attack

Step4. Allocate objs_per_slab-1 objects as pre-alloc objects

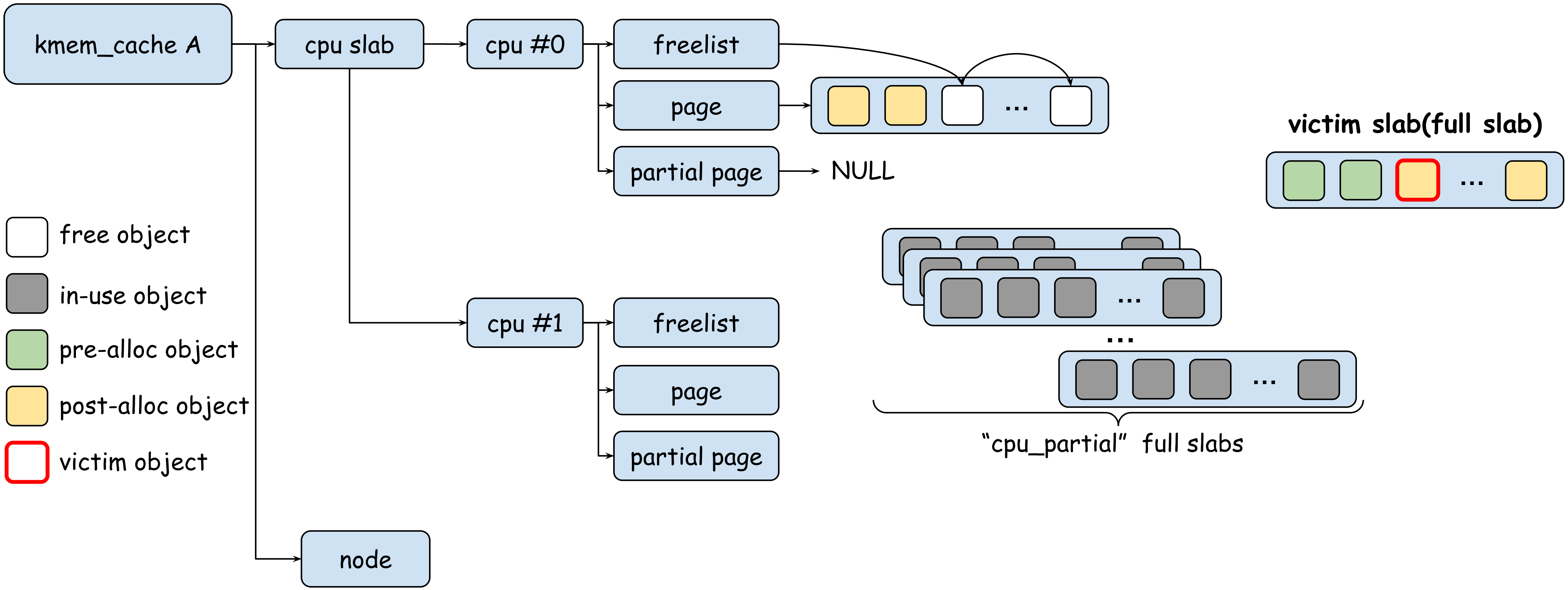
Step5. Allocate the victim object

Step6. Trigger the vulnerability(UAF) to release the victim object



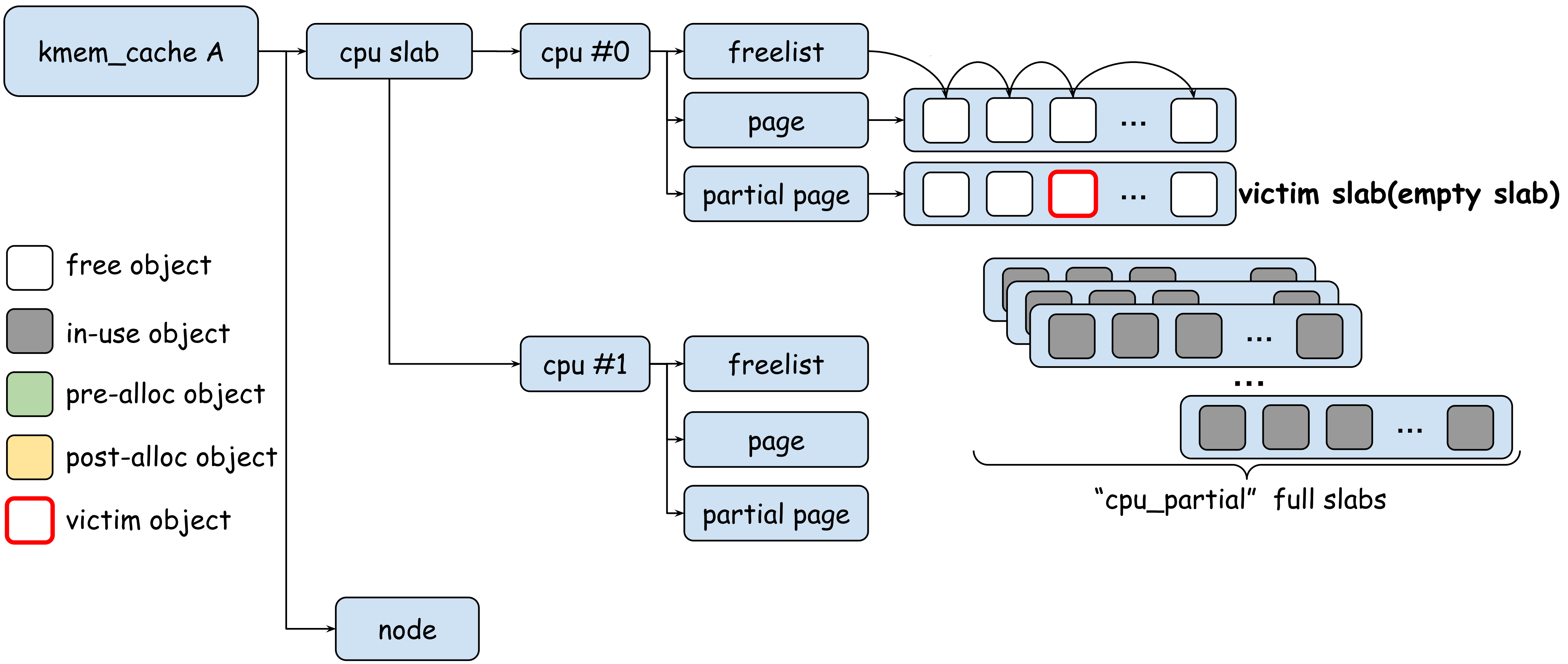
# Common workflow of Cross-cache attack

Step7. Allocate objs_per_slab+1 objects as post-alloc objects



# Common workflow of Cross-cache attack

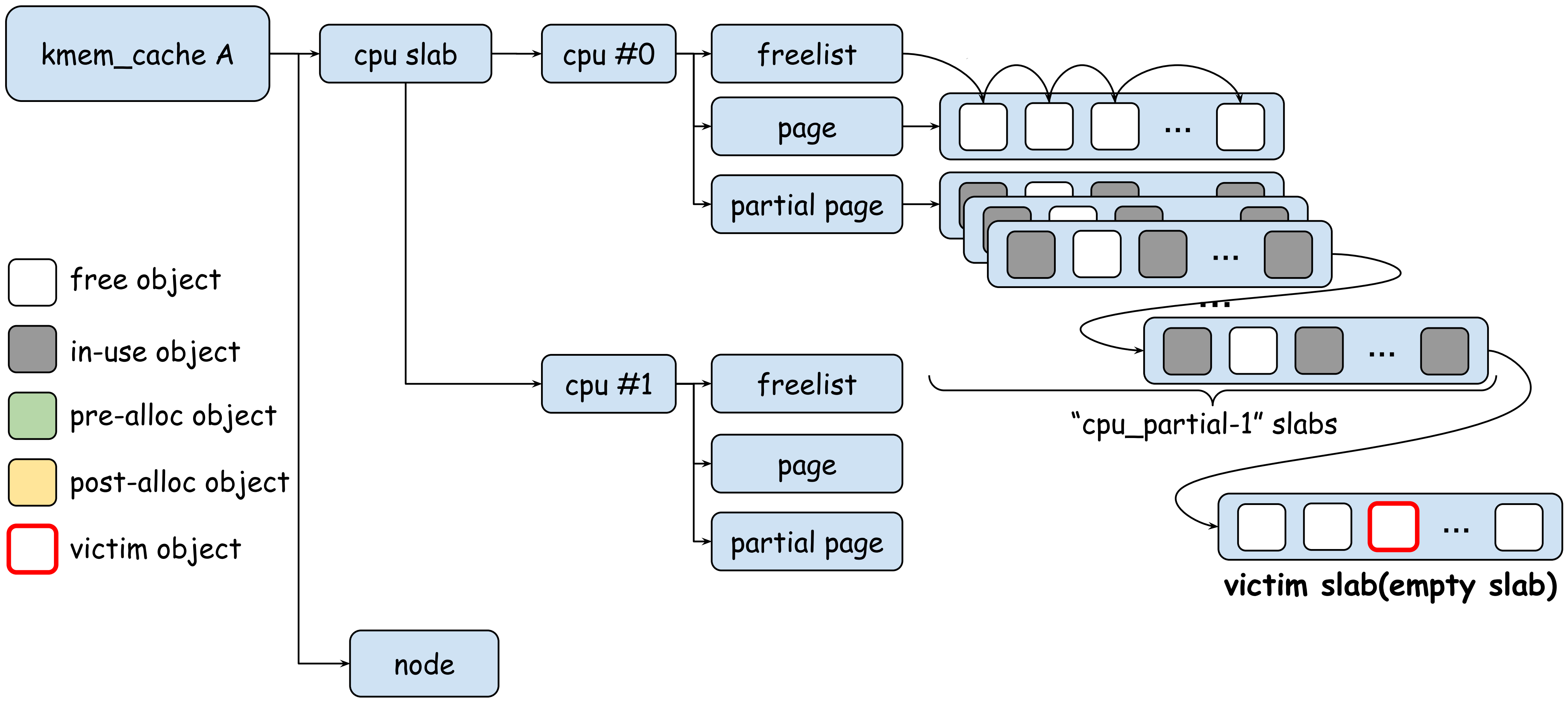
Step8. Release all the pre-alloc and post-alloc objects



# Common workflow of Cross-cache attack

Step9. Free one object per slab from the allocations in Step3

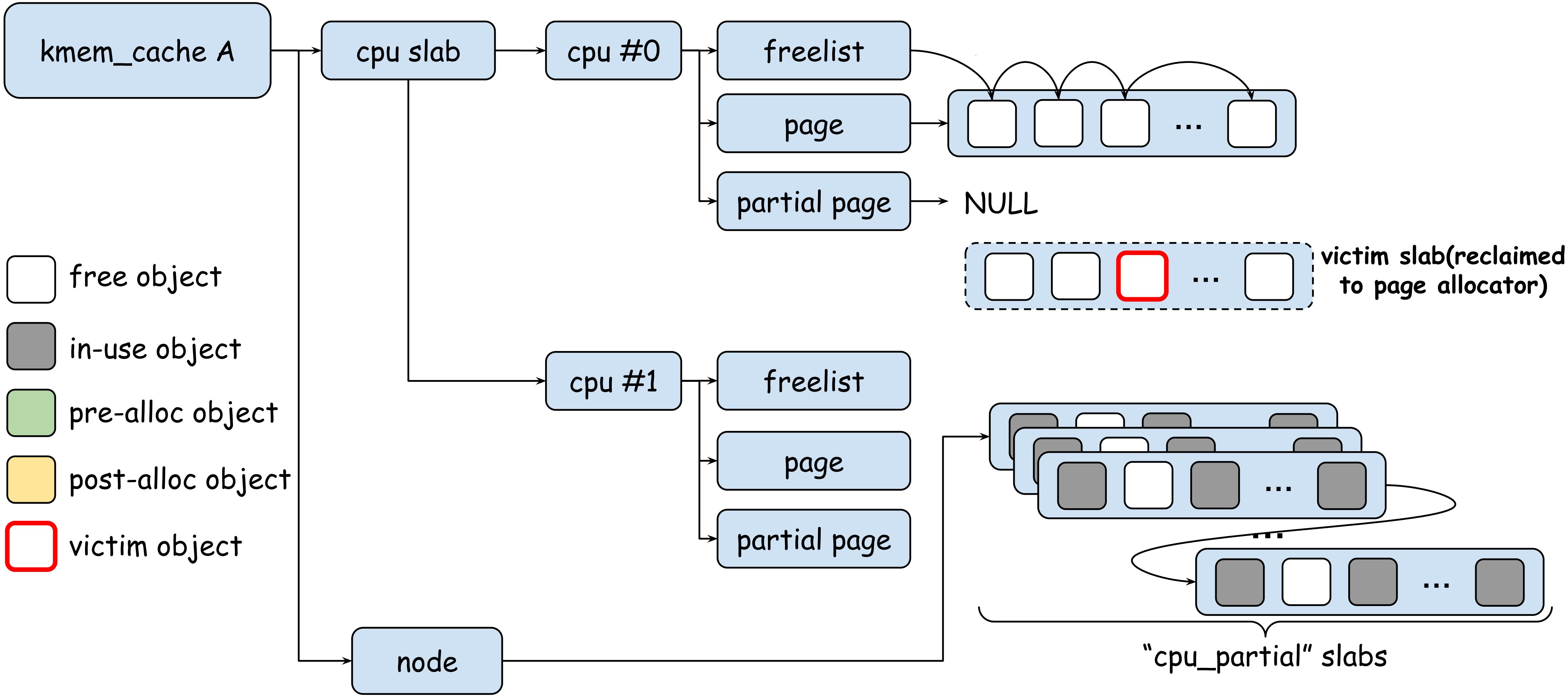
After releasing "cpu_partial – 1" objects:



# Common workflow of Cross-cache attack

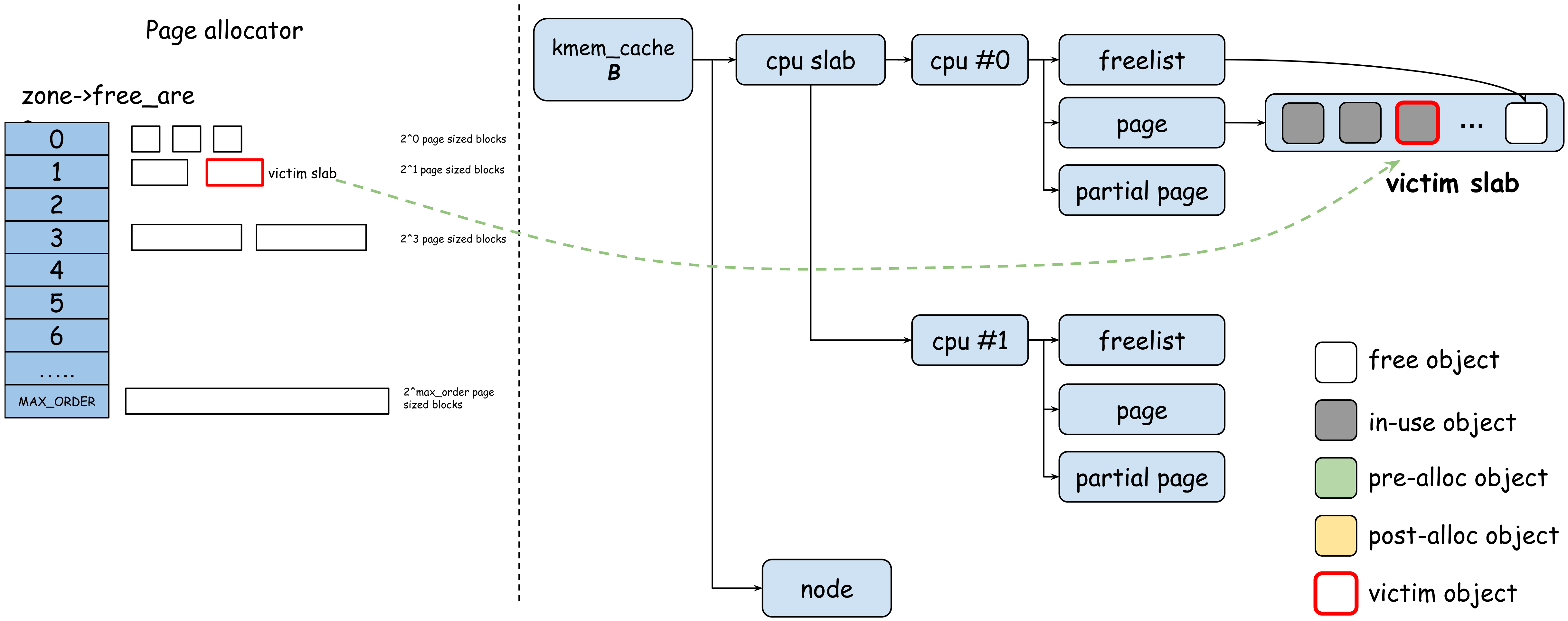
Step9. Free one object per slab from the allocations from Step3

After releasing one more object, the flushing for cpu partial list gets triggered:



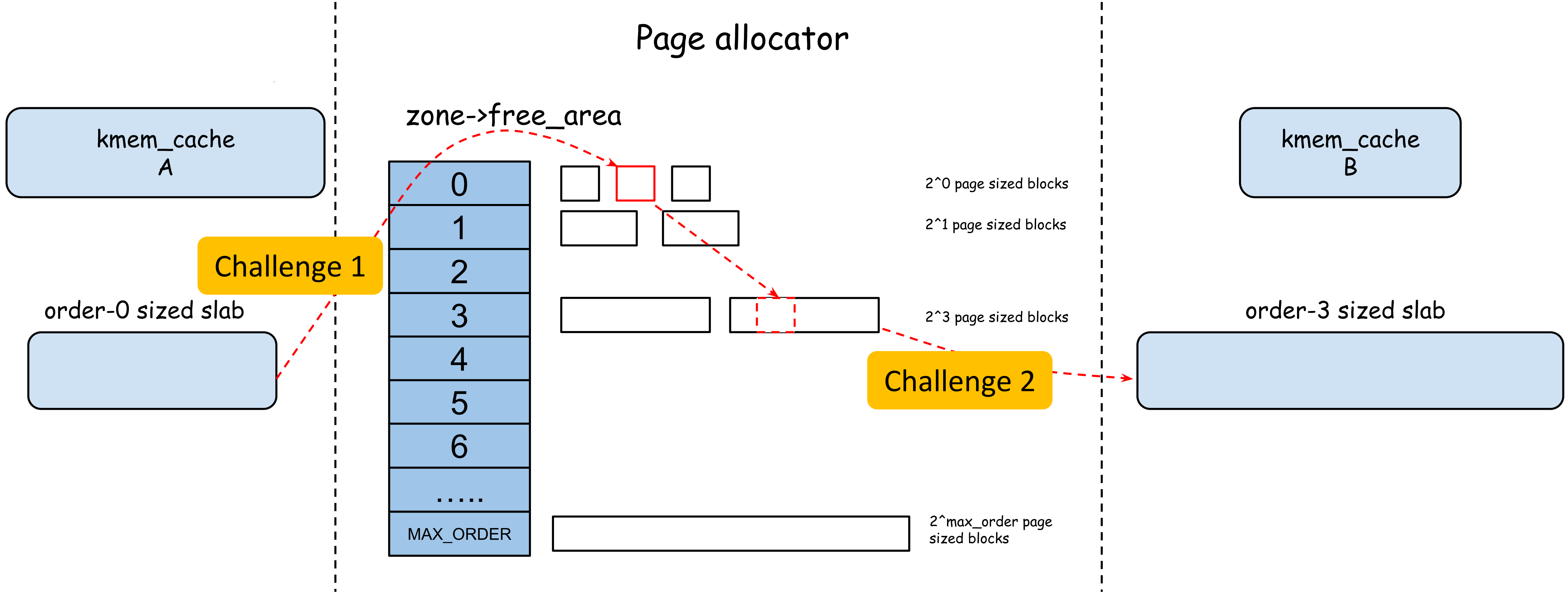
# Common workflow of Cross-cache attack

Step10. Heap spray with object B to occupy the victim slab, victim **object A** gets reallocated as **object B**




Step11. Construct primitives for privilege escalation

# Challenges in Cross-cache attack

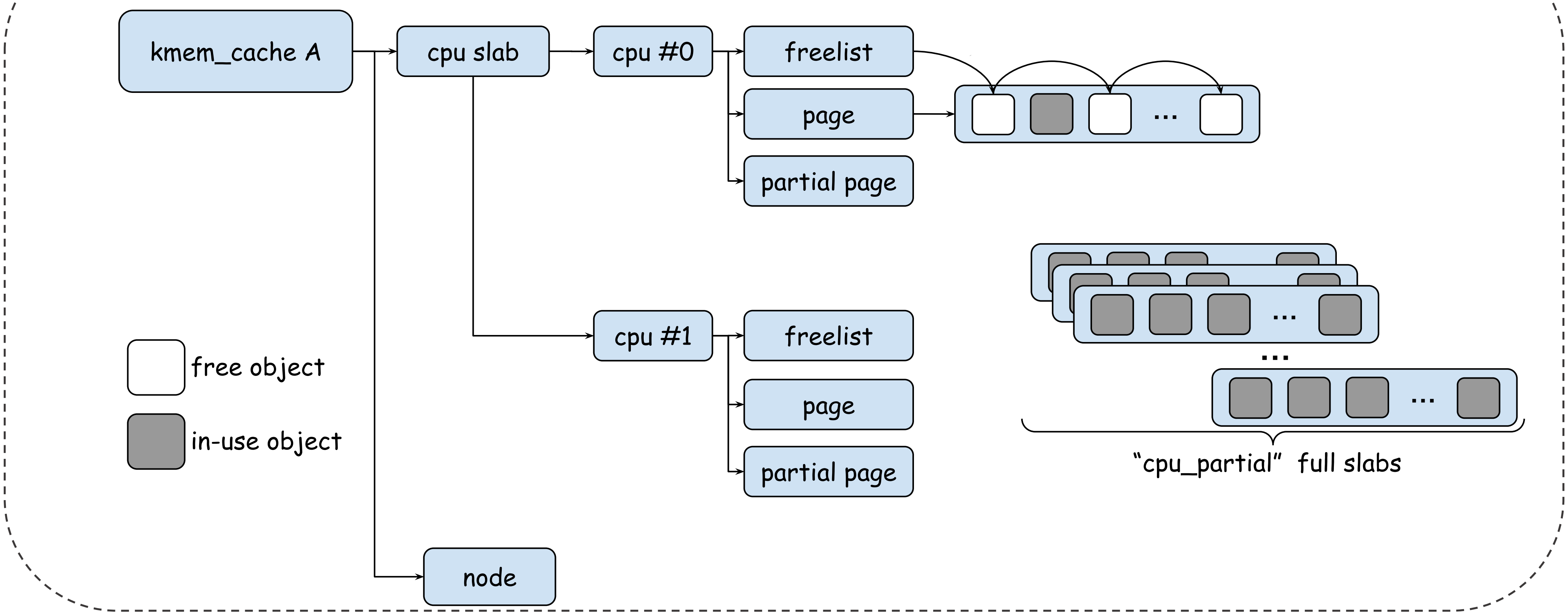



- **Challenge 1**: How to discard the victim slab under a constrained allocation primitive
- **Challenge 2**: How to make high-order slab reuse the low-order slab deterministically

# Challenges in Cross-cache attack

**Challenge 1**: How to discard the victim slab under a constrained allocation primitive

Step 3. Allocate around **objs_per_slab** * (1+**cpu_partial**) objects




This step requires us:

- Allocate a large number of objects
- Keep this large number of objects unreleased for a while

# Challenges in Cross-cache attack

🚫 Allocate a large number of objects

- ❑ Dedicated kmem-cache is becoming a mitigation for cross-cache attack. We can hardly find suitable allocation primitives. The known mitigations like: CONFIG_RANDOM_KMALLOC_CACHES, AUTOSLAB
- ❑ Limited system resources
- ❑ Constraints of kernel components

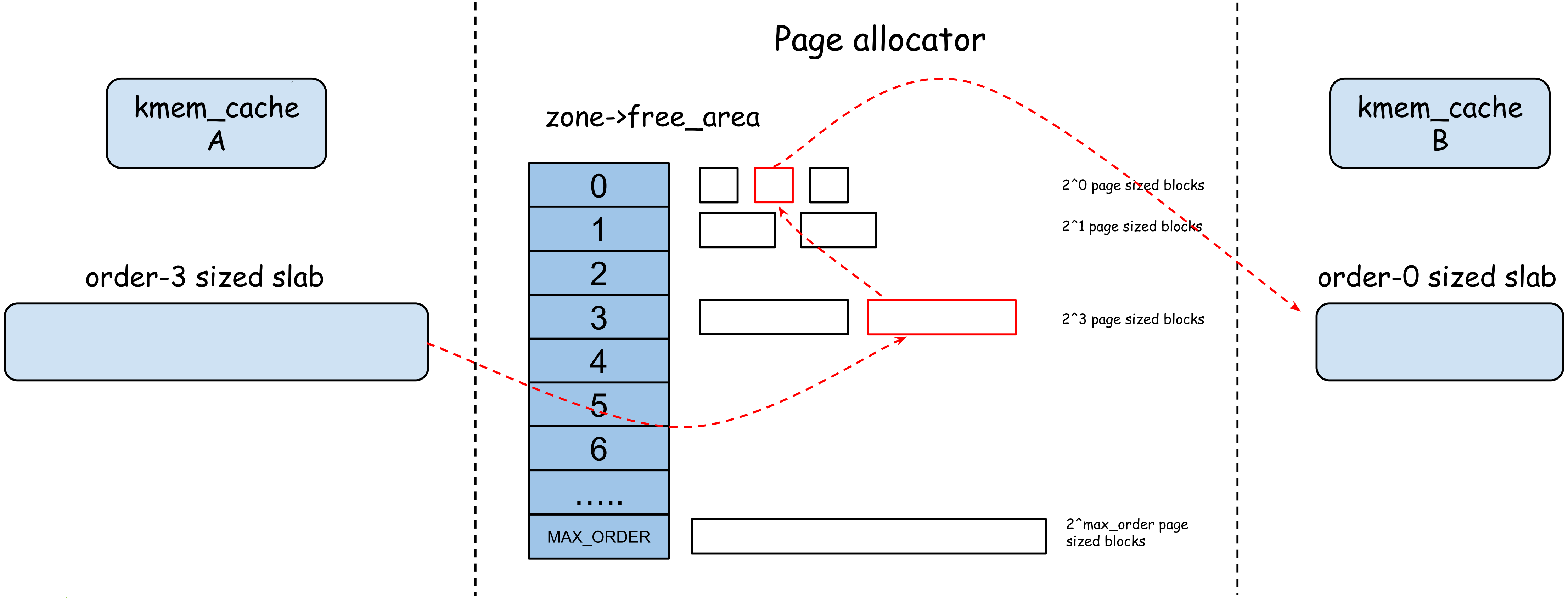
🚫 Keep the large number of objects unreleased for a while

- ❑ Temporary kernel object: gets allocated and then released.

# Challenges in Cross-cache attack

**Challenge 2**: How to make high-order slab reuse the low-order slab deterministically

- order-N pages --> order-M pages, N > M



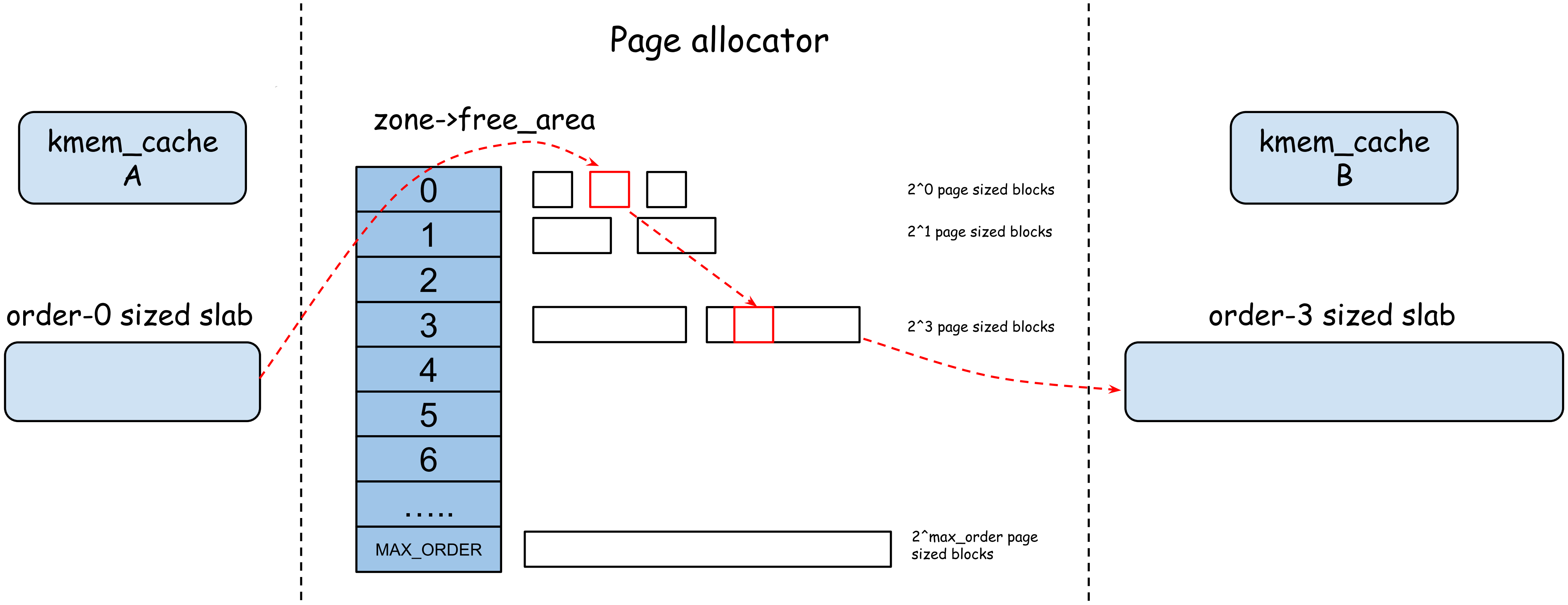

Can be done by allocating tons of object B, order-N pages will definitely be reused as order-M pages.
This may require:

- too many object B, this can be really hard under a limited system resources

# Challenges in Cross-cache attack

**Challenge 2**: How to make high-order slab reuse the low-order slab deterministically

- order-N pages --> order-M pages, N < M




Allocating tons of object B won't help. We need to let order-N pages get compacted into order-M pages, so object B can reuse these order-N pages.

So how? ---- Shaping the heap!

# Advancing Towards a More Effective Cross-Cache Attack

# Advancing Towards a More Effective Cross-Cache Attack

CVE-2023-21400

A NPU issue affected qualcomm 4.14 kernel, can be accessed from unstrusted app, found by Ye Zhang

**Task A(On cpu1)**

```
mutex_lock(&host_ctx->lock);
network = get_network_by_hdl(host_ctx, …,unload->network_hdl);
unload_cmd1 = npu_alloc_network_cmd(host_ctx, 0);
npu_queue_network_cmd(network, unload_cmd1);
mutex_unlock(&host_ctx->lock);
```

**Task B(On cpu2)**

```
mutex_lock(&host_ctx->lock);
network = get_network_by_hdl(host_ctx, …,unload->network_hdl);
unload_cmd2 = npu_alloc_network_cmd(host_ctx, 0);
npu_queue_network_cmd(network, unload_cmd2);
mutex_unlock(&host_ctx->lock);
wait_for_completion_timeout(&unload_cmd2->cmd_done,NW_CMD_TIMEOUT);
mutex_lock(&host_ctx->lock);
npu_dequeue_network_cmd(network, unload_cmd2);
npu_free_network_cmd(host_ctx, unload_cmd2);
free_network(host_ctx, client, network->id);
mutex_unlock(&host_ctx->lock);
```

**unload_cmd1 gets released here!**

20s

```
wait_for_completion_timeout(&unload_cmd1->cmd_done,NW_CMD_TIMEOUT);
mutex_lock(&host_ctx->lock);
npu_dequeue_network_cmd(network, unload_cmd1);
npu_free_network_cmd(host_ctx, unload_cmd1);
free_network(host_ctx, client, network->id);
mutex_unlock(&host_ctx->lock);
```

**UAF or Double free happens!**

# Advancing Towards a More Effective Cross-Cache Attack

CVE-2023-21400)[3]

With the bug, we can:

```
wait_for_completion_timeout(&unload_cmd1->cmd_done,NW_CMD_TIMEOUT);
mutex_lock(&host_ctx->lock);
npu_dequeue_network_cmd(network, unload_cmd1);
npu_free_network_cmd(host_ctx, unload_cmd1);
free_network(host_ctx, client, network->id);
mutex_unlock(&host_ctx->lock);
```

```
static void npu_dequeue_network_cmd(struct npu_network *network,
   struct npu_network_cmd *cmd)
{
    list_del(&cmd->list);
}
```

list_del() primitive

```
static void npu_free_network_cmd(struct npu_host_ctx *ctx,
   struct npu_network_cmd *cmd)
{
   if (cmd->stats_buf)
      kmem_cache_free(ctx->stats_buf_cache, cmd->stats_buf);

   kmem_cache_free(ctx->network_cmd_cache, cmd);
}
```

**Arbitrary kmem_cache_free() primitive**

Double free primitive

[3]:https://i.blackhat.com/EU-23/Presentations/EU-23-Zhang-Attacking-NPUs-of-Multiple-Platforms.pdf

# Advancing Towards a More Effective Cross-Cache Attack

CVE-2023-21400

Victim object:

```
struct npu_network_cmd {
    struct list_head list;
    ...
    struct completion cmd_done;
    /* stats buf info */
    uint32_t stats_buf_size;
    void __user *stats_buf_u;
    void *stats_buf;
    int ret_status;
};
```

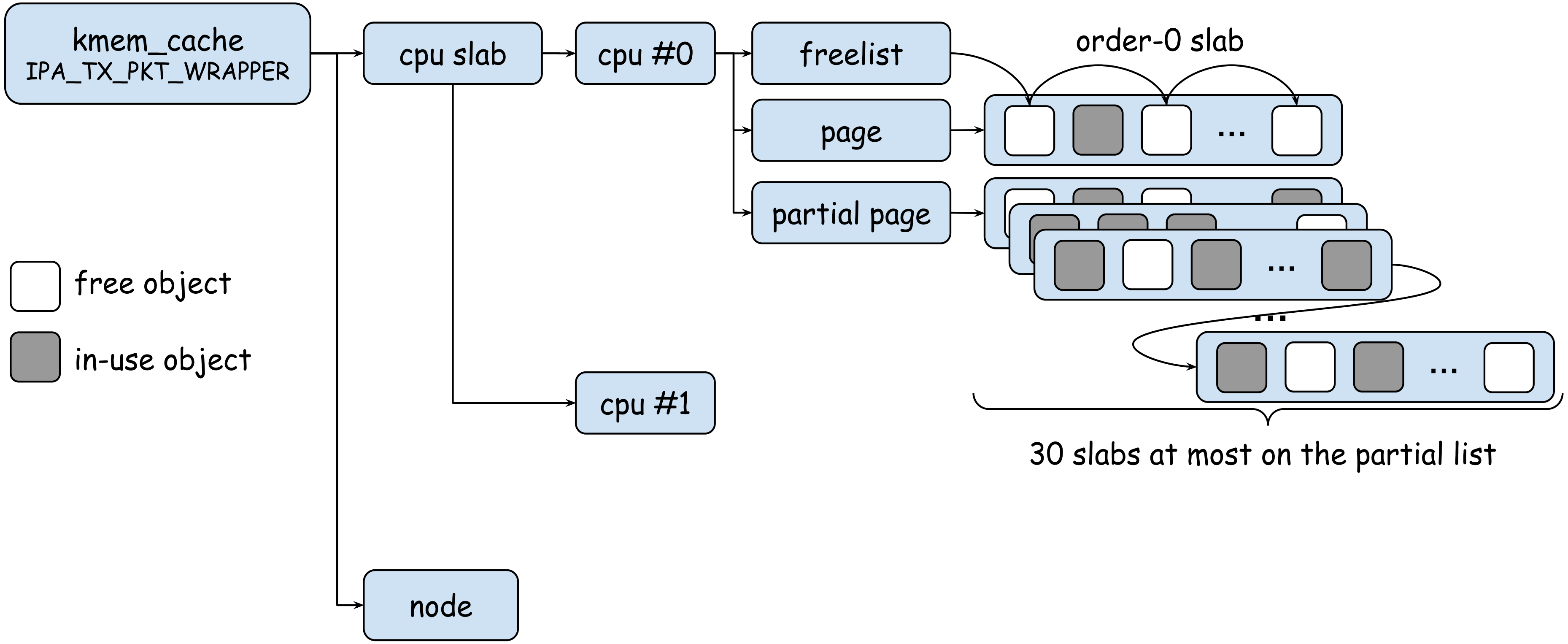
Allocated from a dedicated kmem_cache "IPA_TX_PKT_WRAPPER"

# Advancing Towards a More Effective Cross-Cache Attack

CVE-2023-21400

Allocated from a dedicated kmem_cache "IPA_TX_PKT_WRAPPER"

```
x1q:/sys/kernel/slab/IPA_TX_PKT_WRAPPER # cat order
0
x1q:/sys/kernel/slab/IPA_TX_PKT_WRAPPER # cat object_size
104
x1q:/sys/kernel/slab/IPA_TX_PKT_WRAPPER # cat objs_per_slab
39
x1q:/sys/kernel/slab/IPA_TX_PKT_WRAPPER # cat cpu_partial
30
```

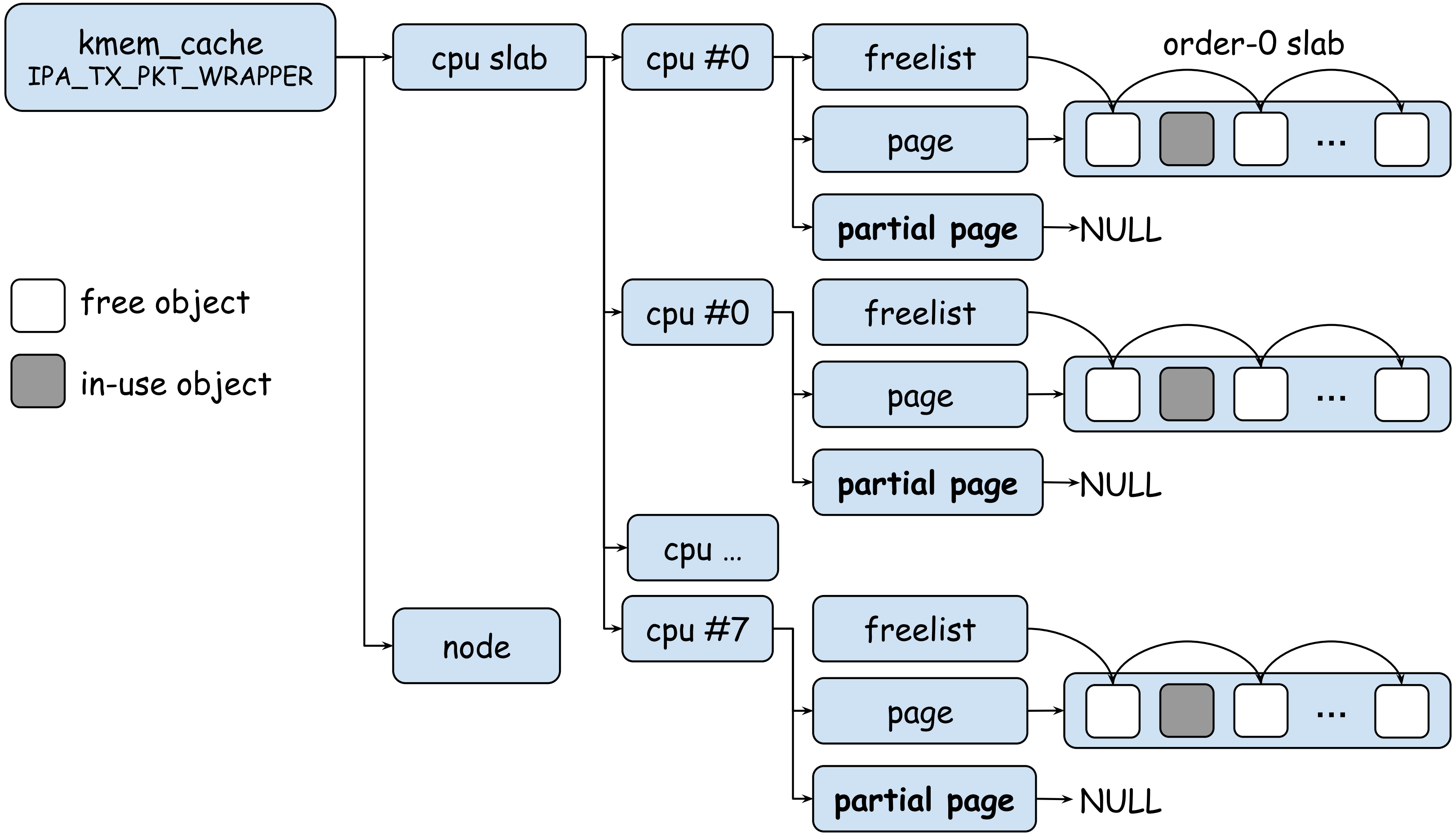


# Advancing Towards a More Effective Cross-Cache Attack

CVE-2023-21400

Allocated from a dedicated kmem_cache "IPA_TX_PKT_WRAPPER"

```
x1q:/sys/kernel/slab/IPA_TX_PKT_WRAPPER # cat slabs_cpu_partial
0(0)
```

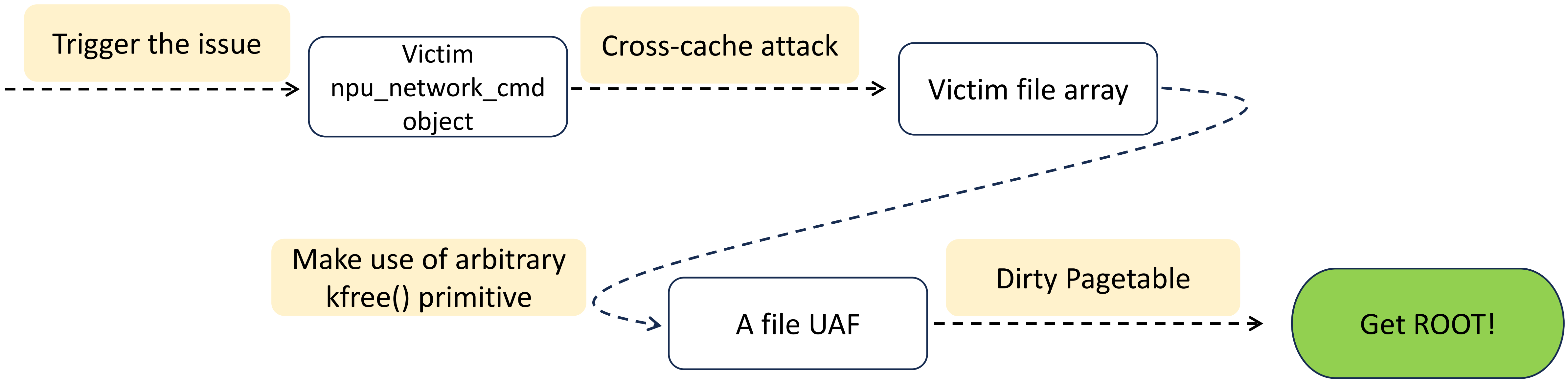



Clean and inactive kmem_cache

# Advancing Towards a More Effective Cross-Cache Attack

CVE-2023-21400

Exploitation plan:




Data-only exploitation, woohoo!

But the cross cache is known for the unstable...

# Advancing Towards a More Effective Cross-Cache Attack

Step1. Trigger the issue

Step2. Cross-cache attack: cross from kmem_cache "IPA_TX_PKT_WRAPPER"  to file_array(kmalloc-8k)

kmem_cache "IPA_TX_PKT_WRAPPER": order-0 slab

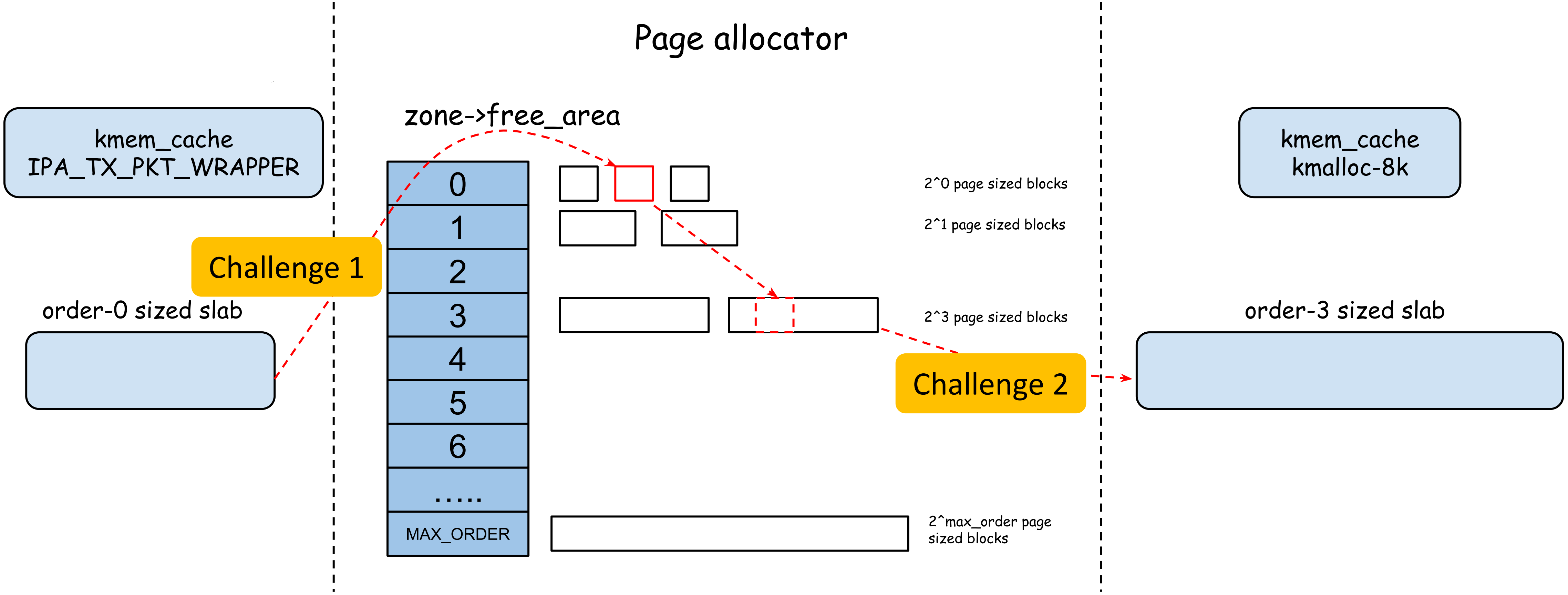
file_array: allocated from kmem_cache "kmalloc-2k" ~ "kmalloc-8k" , all are order-3 slab

```
static struct fdtable * alloc_fdtable(unsigned int nr)
{
    struct fdtable *fdt;
    void *data;
 ...
    nr /= (1024 / sizeof(struct file *));
    nr = roundup_pow_of_two(nr + 1);
    nr *= (1024 / sizeof(struct file *));
    ...
    data = kvmalloc_array(nr, sizeof(struct file *),
GFP_KERNEL_ACCOUNT);
    ...
    fdt->fd = data;
   ...
    return fdt;
...
}
```

We choose kmalloc-8k to allocate file array from.

# Advancing Towards a More Effective Cross-Cache Attack

Step2. Cross-cache attack: cross from kmem_cache "IPA_TX_PKT_WRAPPER"  to file_array(kmalloc-8k)




- Challenge 1: How to discard the victim order-0 slab under a constrained allocation primitive
- Challenge 2: How to make order-3 slab reuse the order-0 slab deterministically

# Advancing Towards a More Effective Cross-Cache Attack

Challenge 1: How to discard the victim order-0 slab under a constrained allocation primitive

- ❑ npu_network_cmd object is a temporary likely kernel object: gets allocated and then released

  - MSM_NPU_LOAD_NETWORK_V2
  - MSM_NPU_UNLOAD_NETWORK
  - MSM_NPU_EXEC_NETWORK_V2 (use this later)

```
struct npu_network_cmd *cmd = NULL;
mutex_lock(&host_ctx->lock);
cmd = kmem_cache_zalloc(ctx->network_cmd_cache, GFP_KERNEL);
mutex_unlock(&host_ctx->lock);

wait_for_npu_firmware();

mutex_lock(&host_ctx->lock);
kmem_cache_free(ctx->network_cmd_cache, cmd);
mutex_unlock(&host_ctx->lock);
```

A really constrained allocation primitive:

We can't Allocate a large number of npu_network_cmd objects and keep this large number of objects unreleased for a while.

# Advancing Towards a More Effective Cross-Cache Attack

Challenge 1: How to discard the victim order-0 slab under a constrained allocation primitive

Well, we found another kernel object sharing the same kmem_cache IPA_TX_PKT_WRAPPER because of SLAB Merging:

From msm_cvp driver:

```
struct msm_cvp_frame {
   struct list_head list;
   struct msm_cvp_list bufs;
   u64 ktid;
};
```

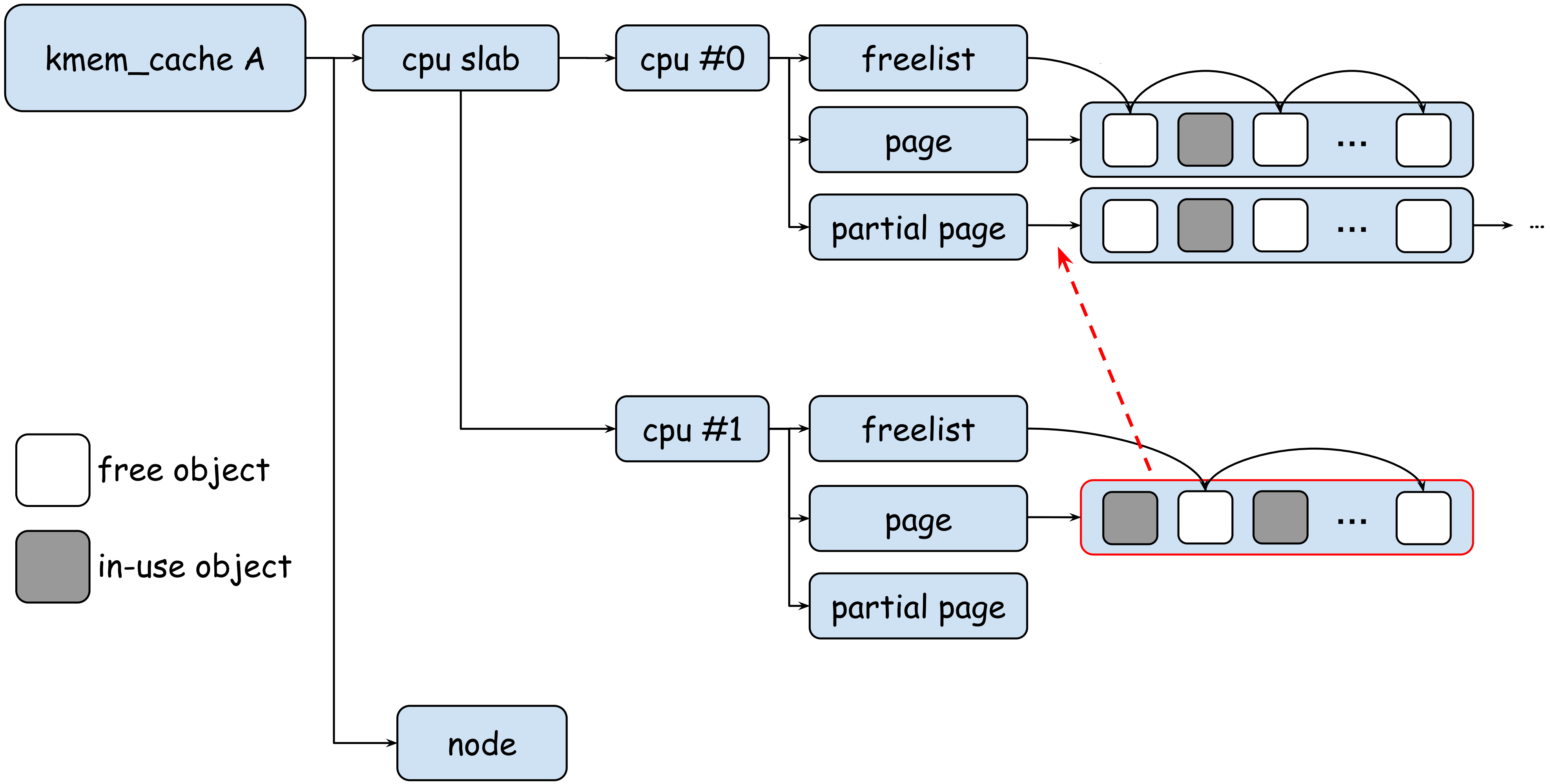
System privilege required to access the driver ☹

So we can't even discard the victim order-0 slab with the old method 😟

# Advancing Towards a More Effective Cross-Cache Attack

Solving Challenge1: Discard the empty slab in a Race way

The slab move primitive: move the cpu slab from one cpu to another cpu's percpu partial list



# Advancing Towards a More Effective Cross-Cache Attack

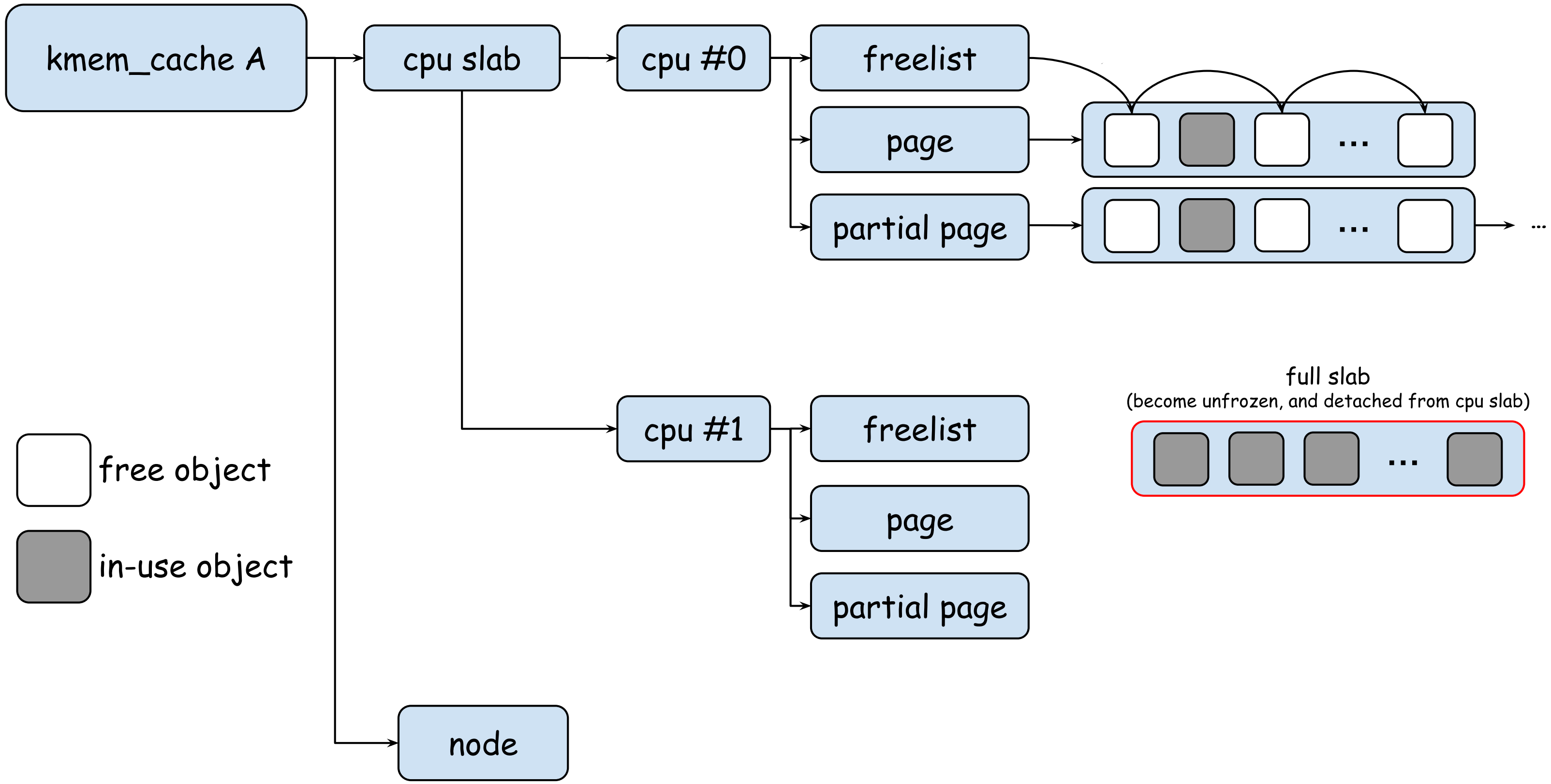
Solving Challenge1: Discard the empty slab in a Race way

The slab move primitive: move the cpu slab from one cpu to another cpu's percpu partial list

**Example**:move cpu slab of cpu#1 into the percpu parital list of cpu#0

**Step1**. Pin the task on cpu#1

**Step2**. Make cpu slab of cpu#1 full by allocating OBJS_PER_SLAB objects



# Advancing Towards a More Effective Cross-Cache Attack

Solving Challenge1: Discard the empty slab in a Race way

The slab move primitive: move the cpu slab from one cpu to another cpu's percpu partial list

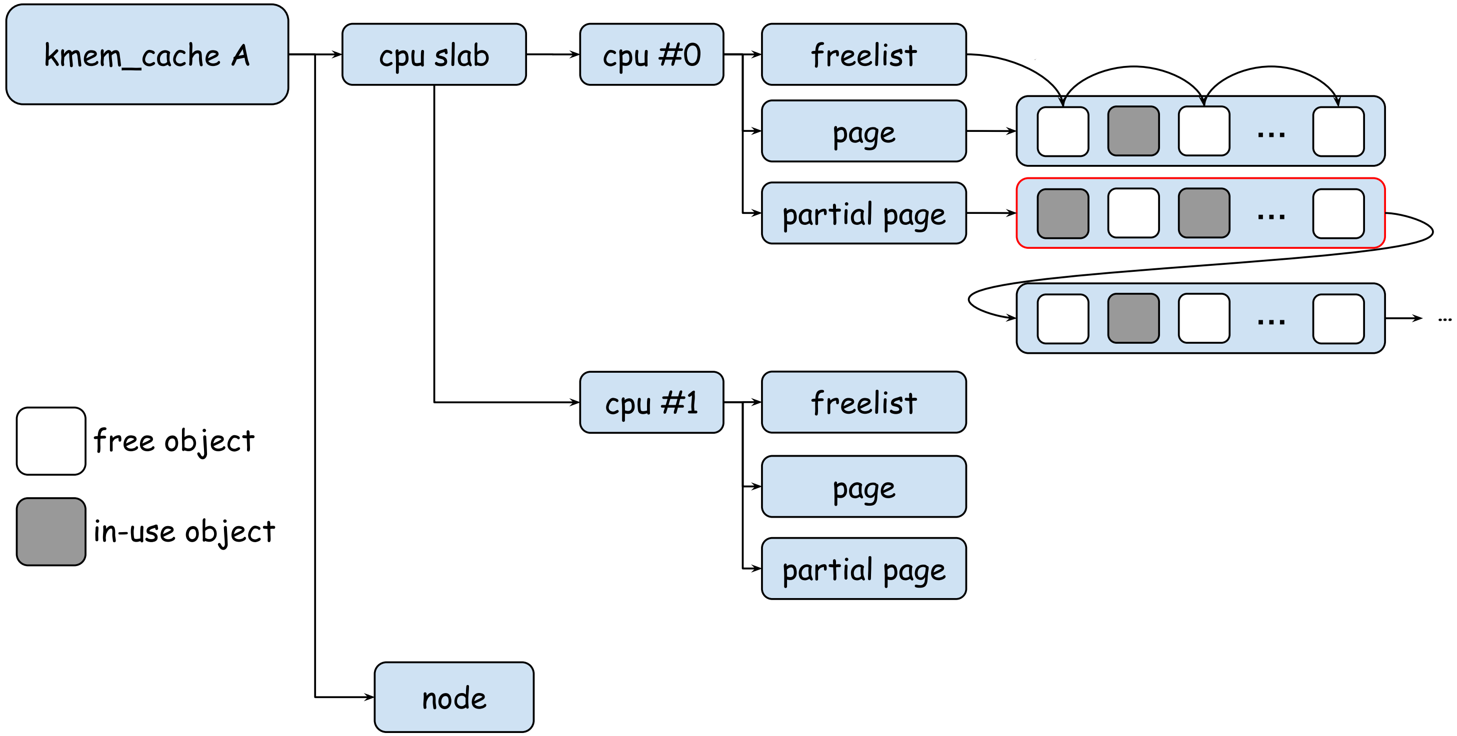
**Example**:move cpu slab of cpu#1 into the percpu parital list of cpu#0

**Step1**. Pin the task on cpu#1

**Step2**. Let cpu slab of cpu#1 become full by allocating OBJS_PER_SLAB objects

**Step3**. Pin the task on cpu#0

**Step4**. Release all the objects allocated in step2. The "slab move" happens （The move would happen when the first object of the full slab get released）

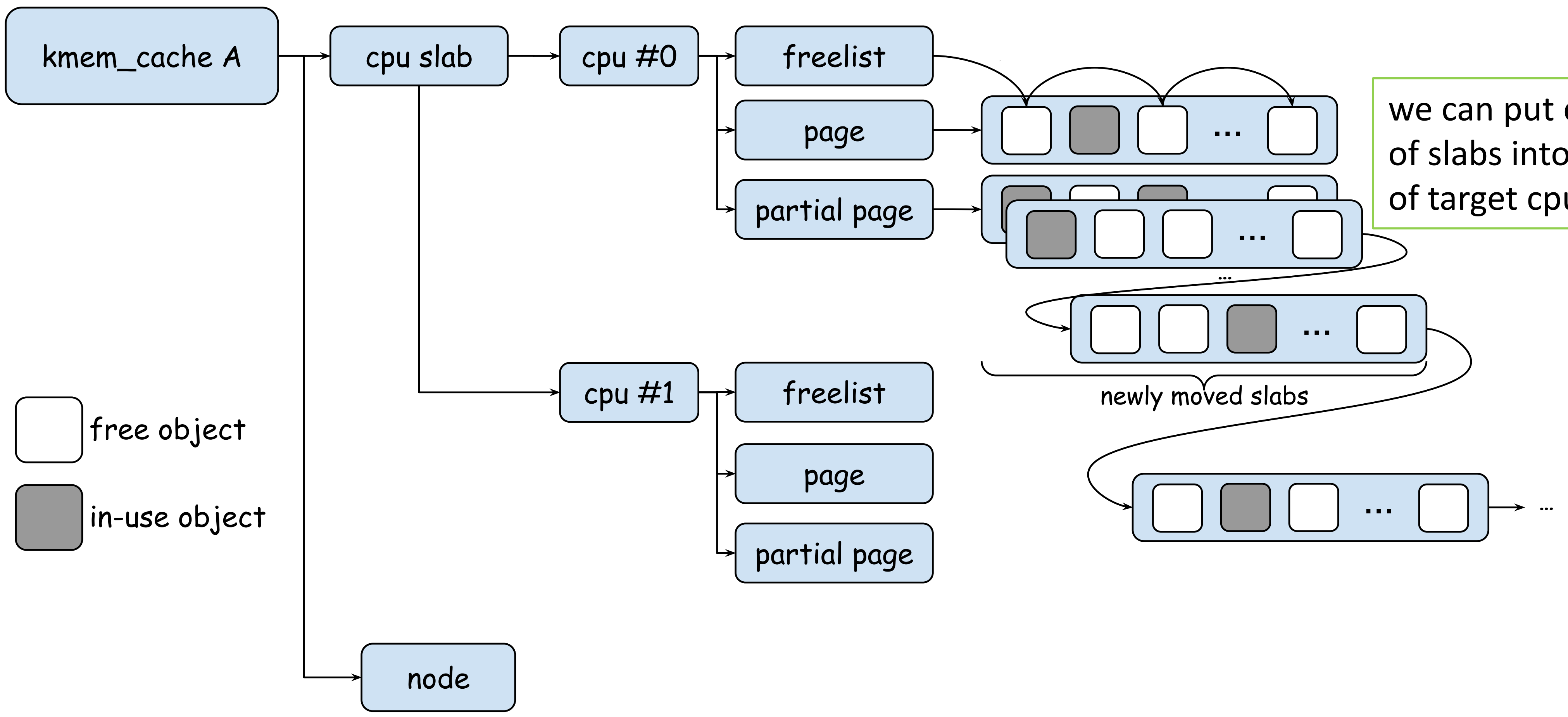



With the help of slab move primitive, we can put one more slab into the cpu partial list of target cpu by allocating OBJS_PER_SLAB objects at most!

# Advancing Towards a More Effective Cross-Cache Attack

Solving Challenge1: Discard the empty slab in a Race way

Repeat the slab move primitive

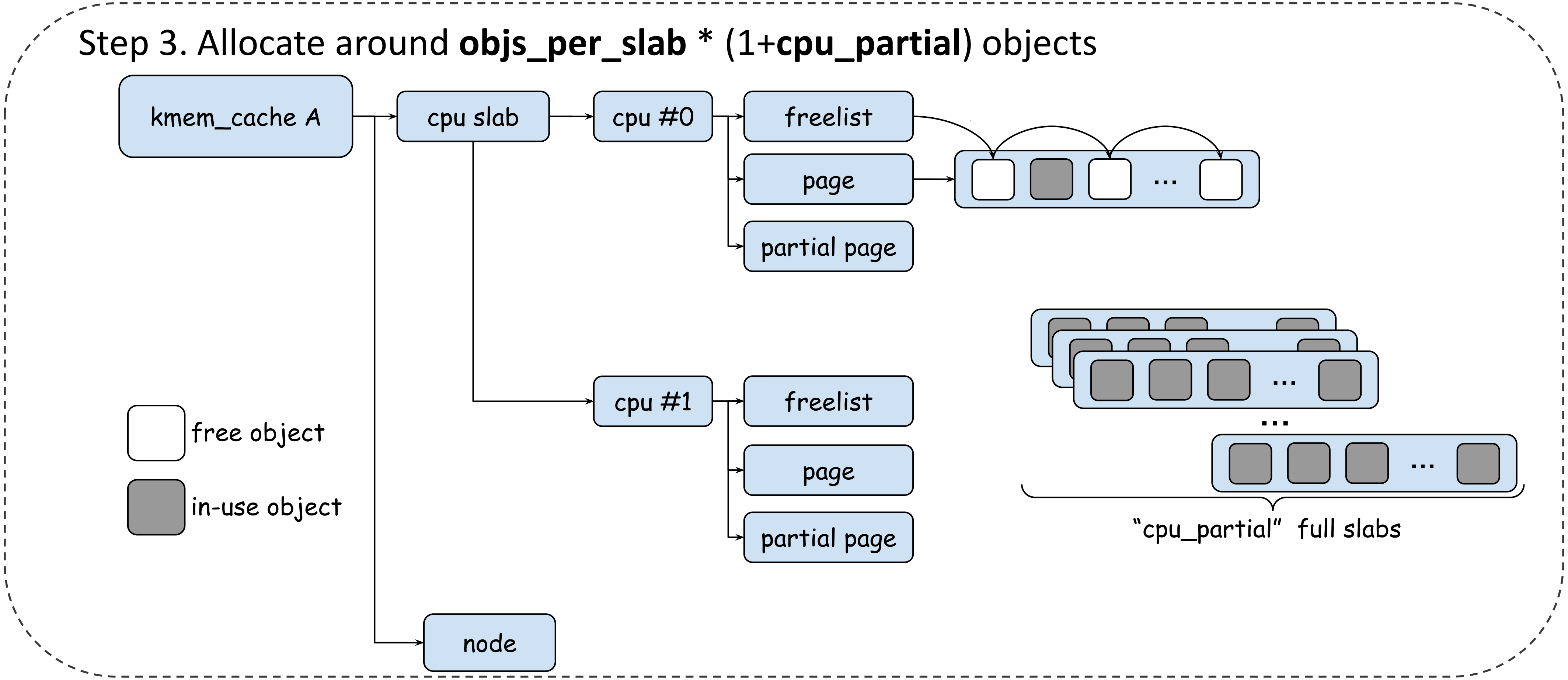



we can put controllable number of slabs into the percpu partial list of target cpu

# Advancing Towards a More Effective Cross-Cache Attack

Solving Challenge1: Discard the empty slab in a Race way

By this new way of putting slabs into the percpu partial list, we can remove the Step3 in common workflow of cross-cache attack, and replace the step9 with "repeating slab move primitive"



# Advancing Towards a More Effective Cross-Cache Attack

Solving Challenge1: Discard the empty slab in a Race way

Repeating slab move pritimive helps us accomplish discarding of victim slab under a very constrained allocation of objects:

Ideally, we can finish the attack with **only OBJS_PER_SLAB objects**!

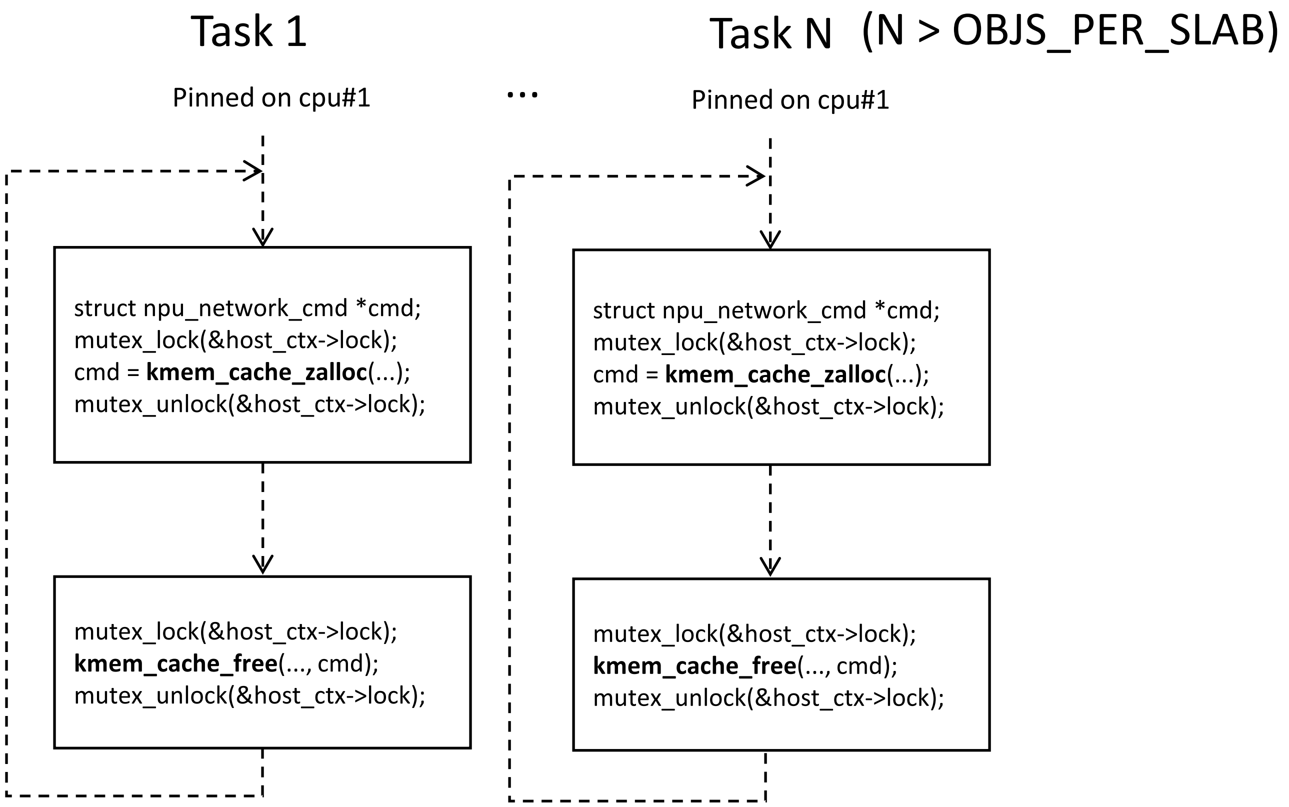
However, it's still not good enough for the issue:

We only have the ability to allocate **one** npu_network_cmd object and hold it for a very short time 😟

# Advancing Towards a More Effective Cross-Cache Attack

Solving Challenge1: Discard the empty slab in a Race way

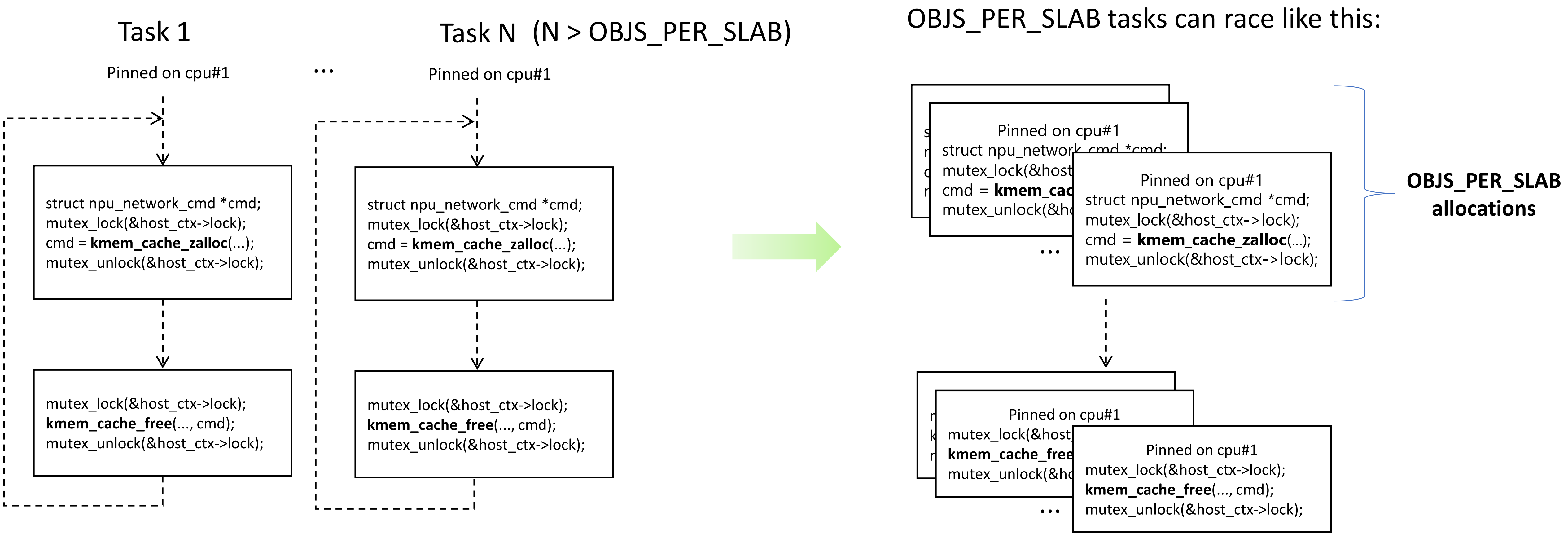
Race style slab move primitive:



# Advancing Towards a More Effective Cross-Cache Attack

Solving Challenge1: Discard the empty slab in a Race way

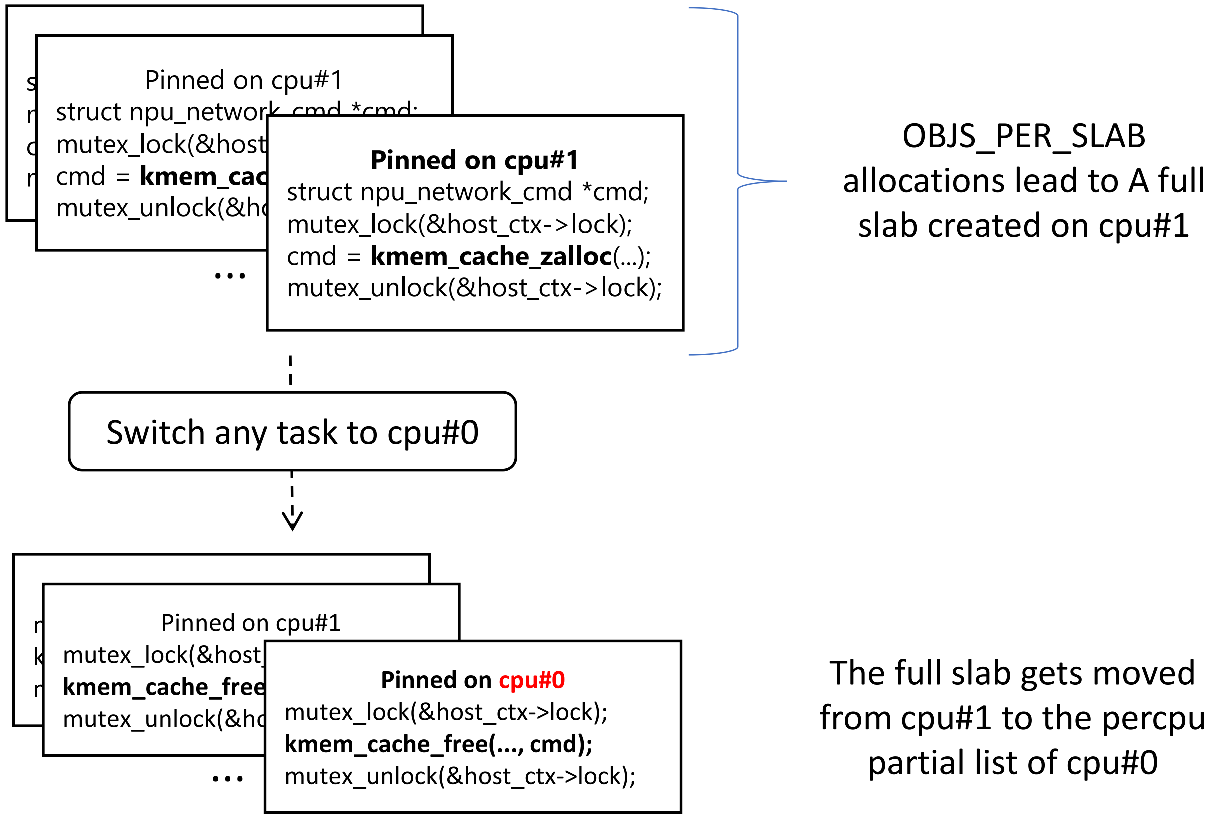
Race style slab move primitive:



# Advancing Towards a More Effective Cross-Cache Attack

Solving Challenge1: Discard the empty slab in a Race way

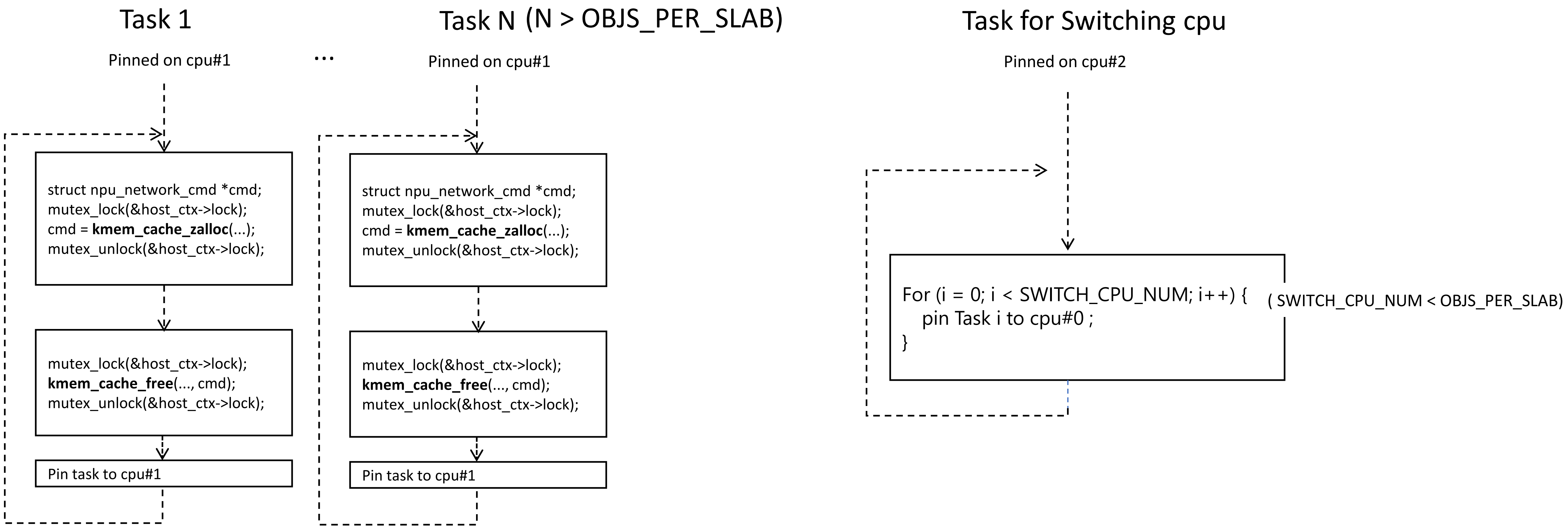
Race style slab move primitive:



# Advancing Towards a More Effective Cross-Cache Attack

Solving Challenge1: Discard the empty slab in a Race way

Model for race style slab move primitive:




(Usually race condition blocks us from exploitation, but this time it helps us)

# Advancing Towards a More Effective Cross-Cache Attack

Solving Challenge1: Discard the empty slab in a Race way

Race style slab move primitive

By adjusting:
- The number of race tasks
- SWITCH_CPU_NUM
- Race time
- Maybe some time window expanding technique ?

→ Move a relatively stable number of slabs into the percpu parital list of cpu#0

Will there be some side effects for the original percpu slabs of cpu#0 ?

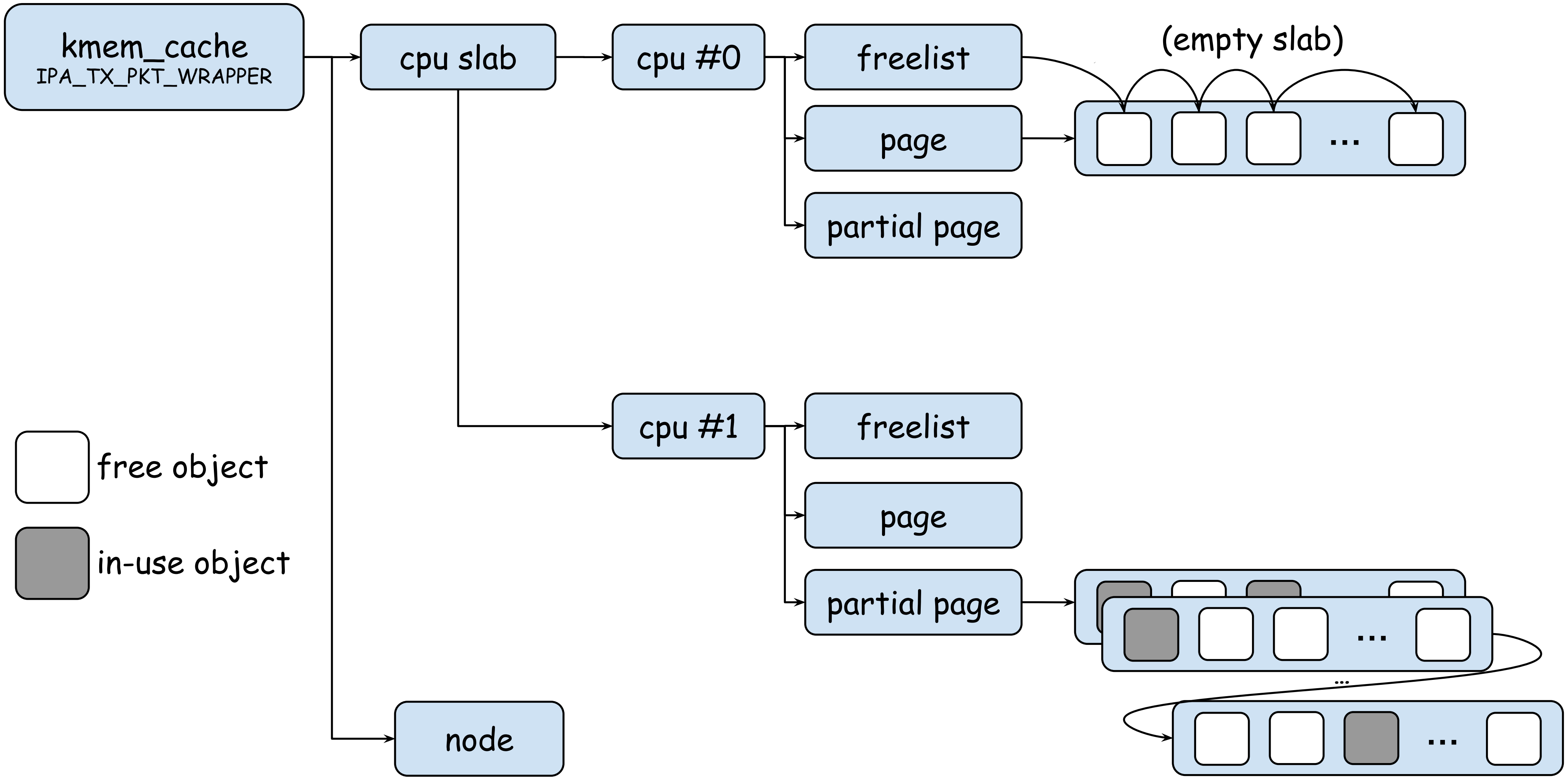
Not really. In the worst case, we might allocate SWITCH_CPU_NUM objects on cpu#0, which won't create a full slab on cpu#0, so:
- If any of these objects gets released on cpu#0, no slab move would happen because we are the same cpu
- If any of these objects gets released on cpu#1, no slab move would happen because the slab is not full

✓ With the race style slab move primitive, we can easily all add enough slabs into the percpu partial list, and then succeed in reclaiming the empty slab with a really constrained allocation.

# Advancing Towards a More Effective Cross-Cache Attack

## The new optimized workflow of cross-cache attack for the issue

Step1. Defragmentation with race style slab move primitive, a **new** slab will be created:

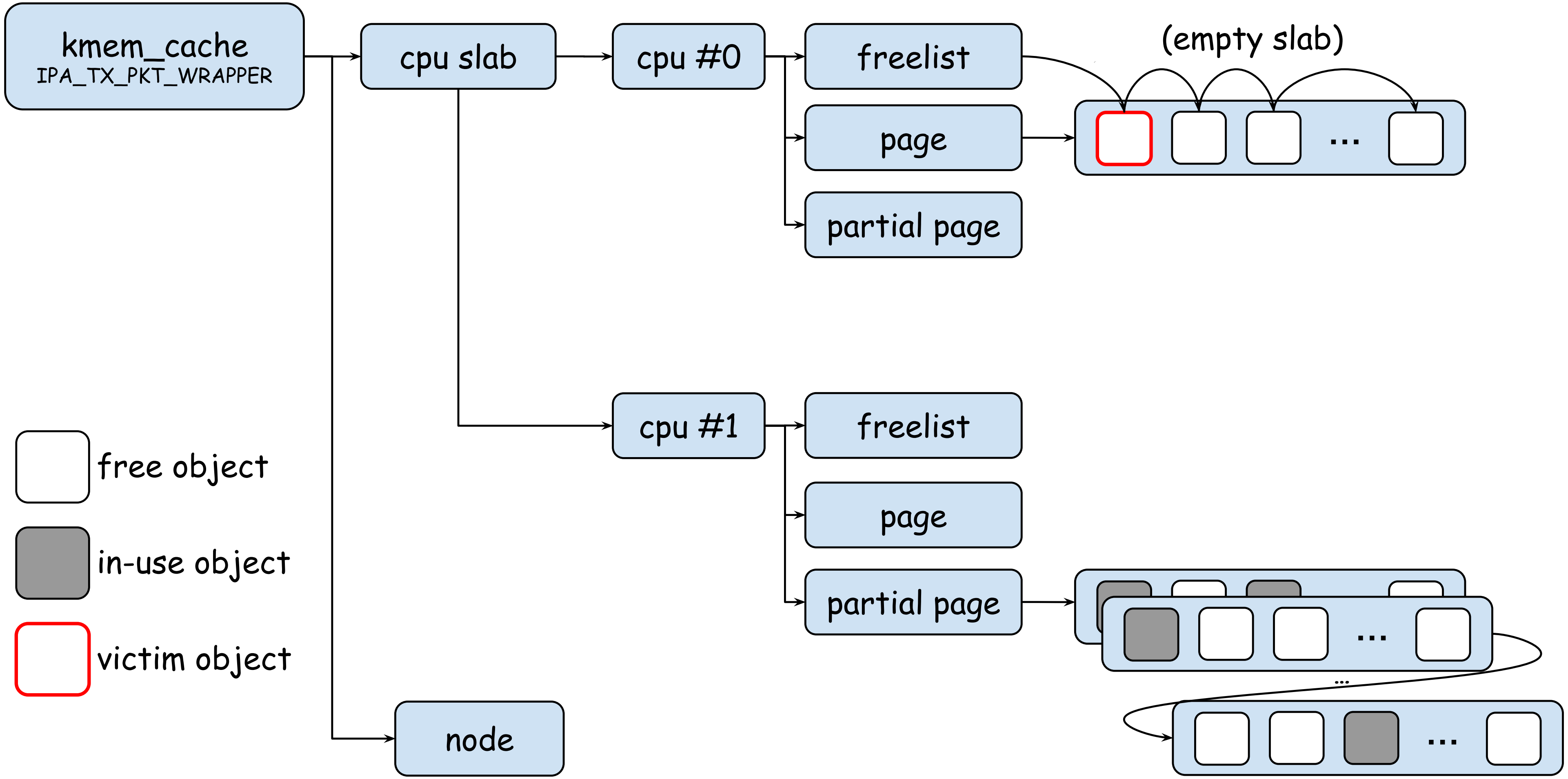


# Advancing Towards a More Effective Cross-Cache Attack

## The new optimized workflow of cross-cache attack for the issue

Step2. Allocate the victim object

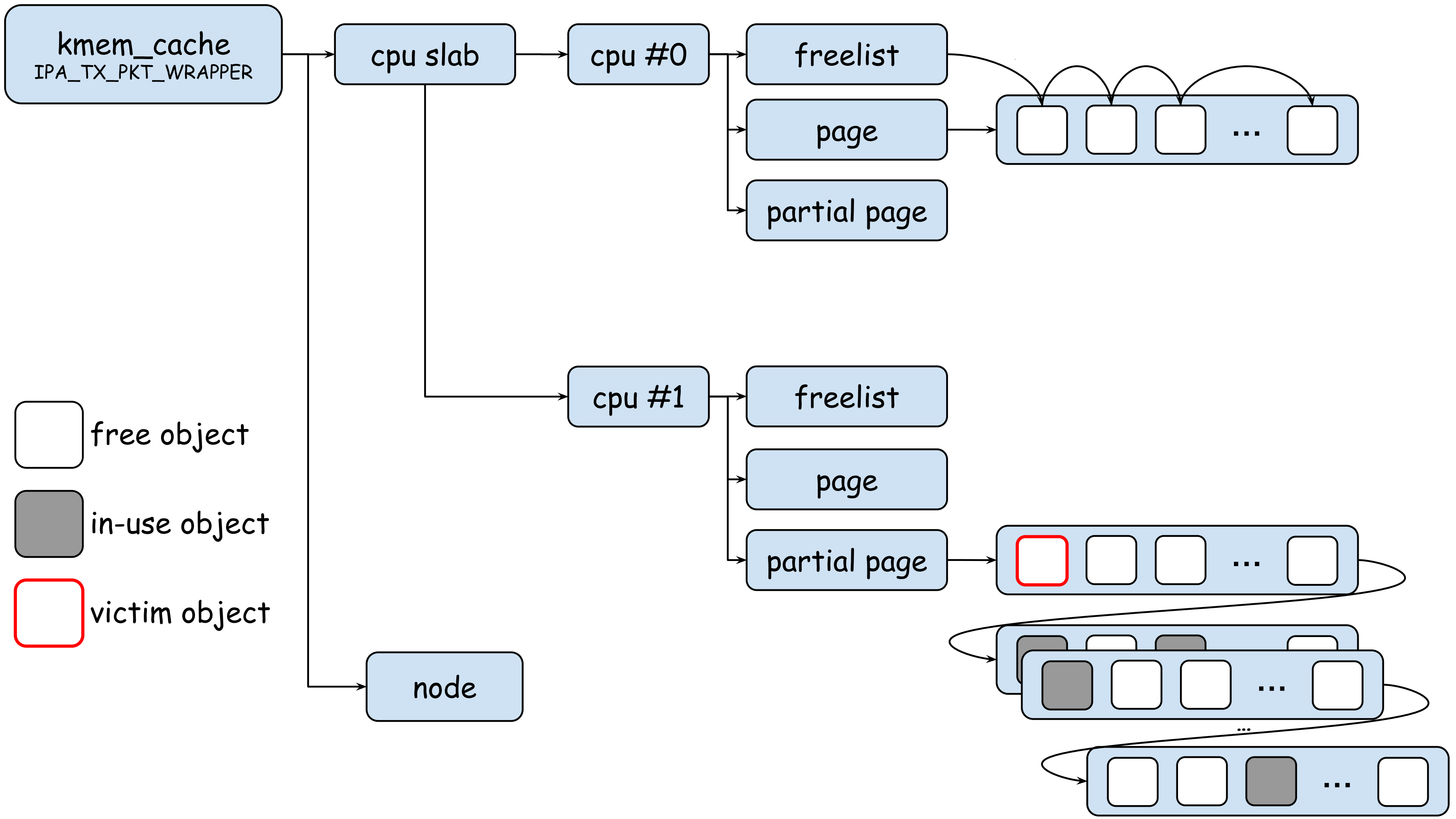
Step3. Trigger the vulnerability(UAF) to release the victim object



# Advancing Towards a More Effective Cross-Cache Attack

## The new optimized workflow of cross-cache attack for the issue

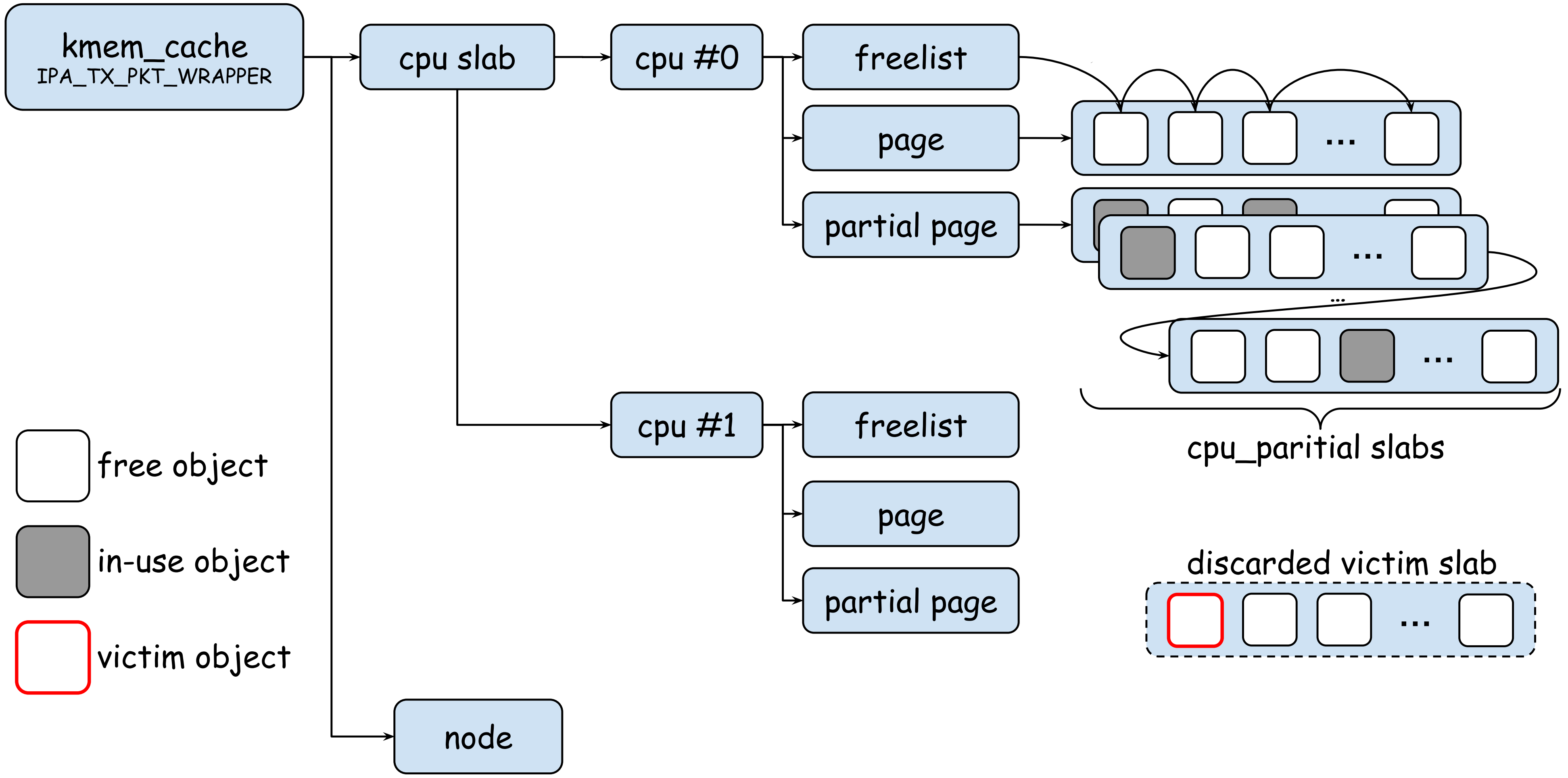
Step4. Move the victim slab to the percpu partial list of cpu#1. Don't trigger the flushing of percpu partial list



# Advancing Towards a More Effective Cross-Cache Attack

## The new optimized workflow of cross-cache attack for the issue

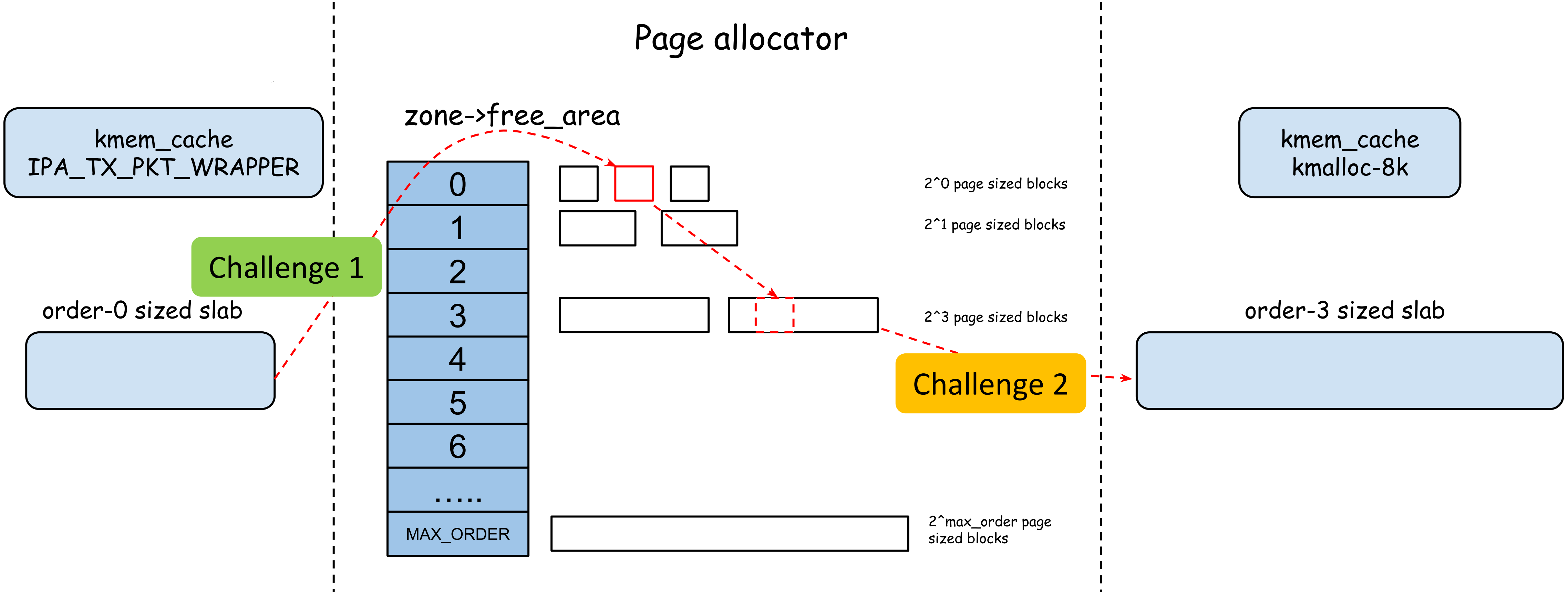
Step 5: move the victim slab from the percpu partial list of cpu#1 to cpu#0. Trigger flushing of percpu partial list of cpu#0




Step 6: Heap spray with file array to occupy the victim slab

# Advancing Towards a More Effective Cross-Cache Attack

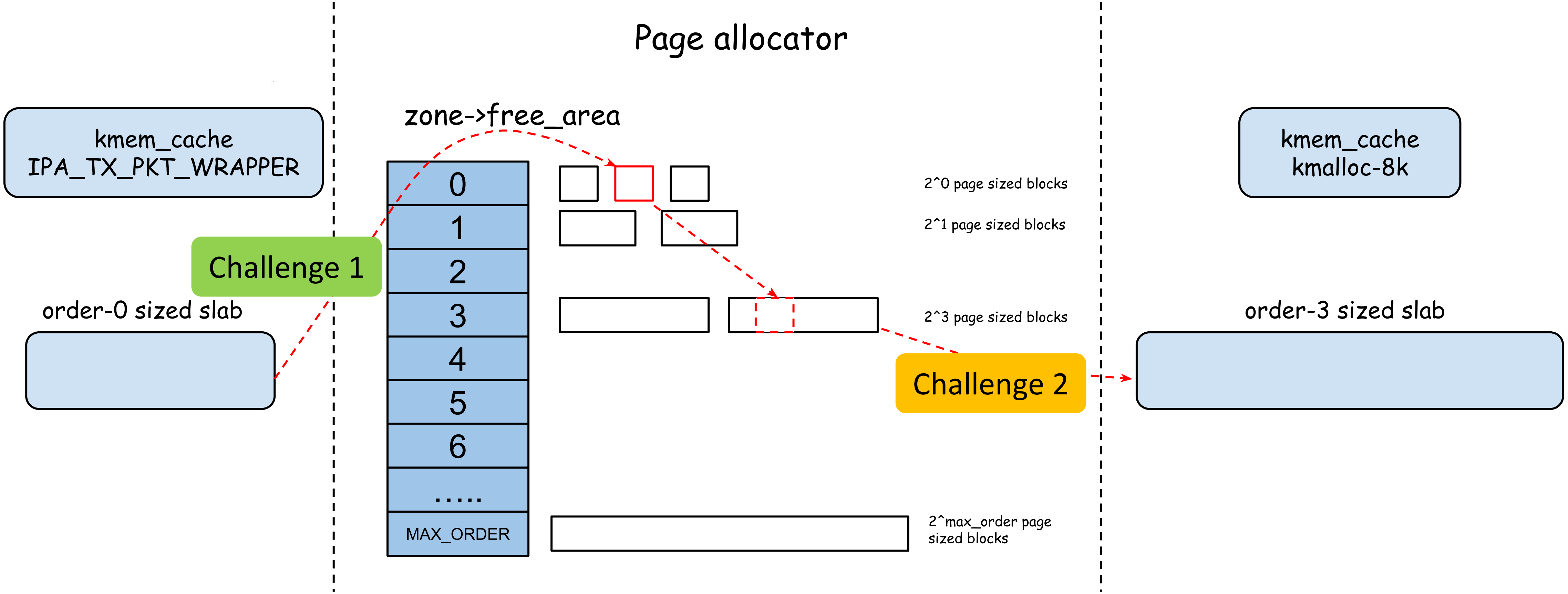
Step2. Cross-cache attack: cross from kmem_cache "IPA_TX_PKT_WRAPPER"  to file_array(kmalloc-8k)




- Challenge 1: How to discard the victim order-0 slab under a constrained allocation primitive SOLVED!
- Challenge 2: How to make order-3 slab reuse the order-0 slab deterministically

# Advancing Towards a More Effective Cross-Cache Attack

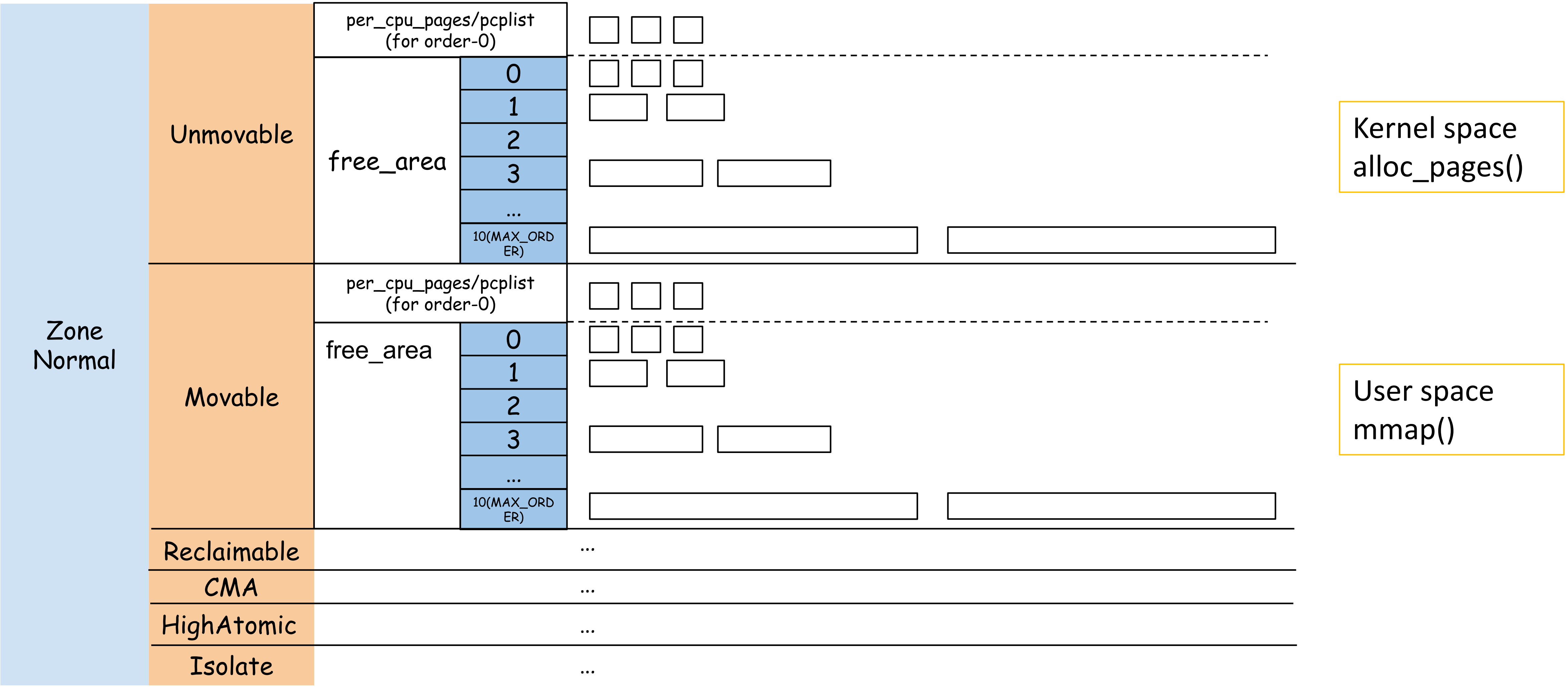
**Challenge 2**: How to make order-3 slab reuse the order-0 slab deterministically



# Advancing Towards a More Effective Cross-Cache Attack

Pre-knowledge for page allocator (based on kernel 4.14)

A simplified view of page allocator for Android devices:(single pgdata & single zone)



# Advancing Towards a More Effective Cross-Cache Attack

Pre-knowledge for page allocator (based on kernel 4.14)

Exported by procfs

/proc/pagetypeinfo (unreadable by untrusted app)

```
x1q:/ # cat /proc/pagetypeinfo
Page block order: 10
Pages per block:  1024

Free pages count per migrate type at order       0      1      2      3      4      5      6      7      8      9     10
Node    0, zone   Normal, type    Unmovable   4828   4818   2414    958    335    112     41      4      0     26     23
Node    0, zone   Normal, type      Movable   4104    516    383    103     34     17      9      5      3      1    169
Node    0, zone   Normal, type  Reclaimable     36     21      4      5      6      2      0      0      0      1      0
Node    0, zone   Normal, type          CMA    399      3      0      0      0      0      0      0      0      0      0
Node    0, zone   Normal, type   HighAtomic      0      0      0      0      0      0      0      0      0      0      0
Node    0, zone   Normal, type      Isolate      0      0      0      0      0      0      0      0      0      0      0

Number of blocks type     Unmovable      Movable  Reclaimable          CMA   HighAtomic      Isolate
Node 0, zone   Normal          1018         1819           39          112            0            0
```

# Advancing Towards a More Effective Cross-Cache Attack

Pre-knowledge for page allocator (based on kernel 4.14)

Exported by procfs

/proc/zoneinfo (unreadable by untrusted app)

```
pages free     269023
      min      3190
      low      52429
      high     55143
      spanned  3144192
      present  3057989
      managed  2714091
      protection: (0, 0)
    nr_free_pages 269023
    nr_zone_inactive_anon 2540
    nr_zone_active_anon 431001
    nr_zone_inactive_file 883667
    nr_zone_active_file 197123
    nr_zone_unevictable 893
    nr_zone_write_pending 18
    nr_mlock     893
    nr_page_table_pages 23417
    nr_kernel_stack 51680
    nr_bounce    0
    nr_zspages   10
    nr_free_cma  20
    nr_free_rbin 0
```

High watermark for zone

```
pagesets
  cpu: 0
            count: 352
            high:  378
            batch: 63
vm stats threshold: 64
  cpu: 1
            count: 345
            high:  378
            batch: 63
vm stats threshold: 64
  cpu: 2
            count: 367
            high:  378
            batch: 63
vm stats threshold: 64
  cpu: 3
            count: 258
            high:  378
            batch: 63
vm stats threshold: 64
  cpu: 4
            count: 326
            high:  378
            batch: 63
```

Current number of order-0 pages

Maxium number of order-0 pages

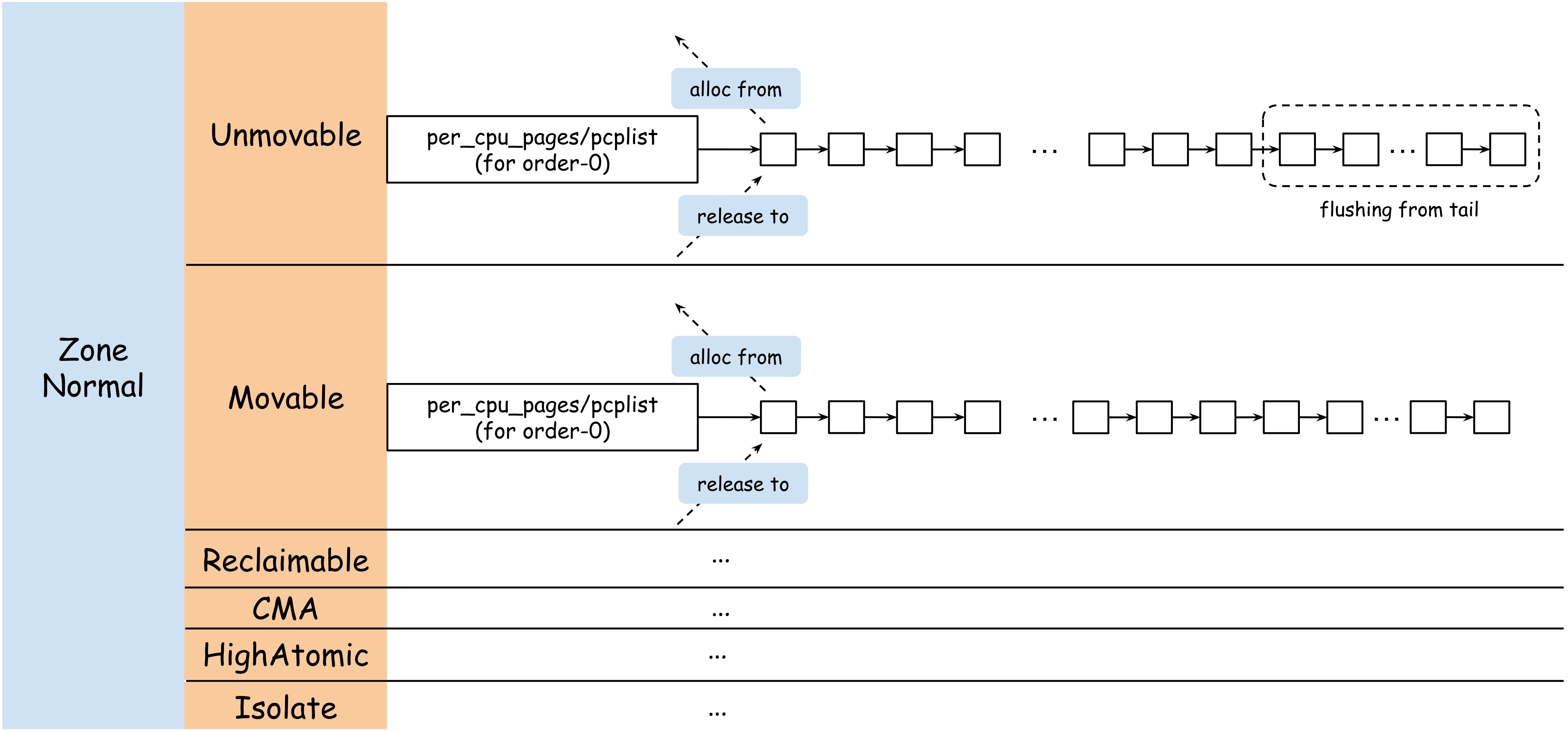
Specific number of order-0 pages for pcplist shrink or bulk

# Advancing Towards a More Effective Cross-Cache Attack

Pre-knowledge for page allocator (based on kernel 4.14)

Charactoristic of pcplist

- Order-0 allocation and releasing will use pcplist first, stack-liked way
- Flushing for the pcplist: flush from tail



# Advancing Towards a More Effective Cross-Cache Attack

Pre-knowledge for page allocator (based on kernel 4.14)

Deterministic page merging:

Page allocator tends to merge low-order pages to high-order pages when low-order pages gets reclaimed into free_area.

```
static inline void __free_one_page(struct page *page,
        unsigned long pfn,
        struct zone *zone, unsigned int order,
        int migratetype)
```

...

```
continue_merging:
    while (order < max_order - 1) {
        ...
        buddy_pfn = __find_buddy_pfn(pfn, order);
        buddy = page + (buddy_pfn - pfn);

        if (!pfn_valid_within(buddy_pfn))
            goto done_merging;
        if (!page_is_buddy(page, buddy, order))
            goto done_merging;
        /*
         * Our buddy is free or it is CONFIG_DEBUG_PAGEALLOC guard page,
         * merge with it and move up one order.
         */
        if (page_is_guard(buddy)) {
            clear_page_guard(zone, buddy, order, migratetype);
        } else {
            list_del(&buddy->lru);
            zone->free_area[order].nr_free--;
            rmv_page_order(buddy);
        }
        combined_pfn = buddy_pfn & pfn;
        page = page + (combined_pfn - pfn);
        pfn = combined_pfn;
        order++;
    }
```

# Advancing Towards a More Effective Cross-Cache Attack

Solving Challenge2: Deterministic heap shaping

Step1: Pin task on cpu#0

Step2: Allocate a specific number of order-0 pages, the specific number is: maxium number of order-0 pages could be in pcplist. Releasing these pages will definitely trigger the flushing or pcplist later.

Choosing the proper kernel component:

Requirements for page allocation:

- Able to allocate a large number of order-0 pages
- Allocated from UNMOVALE free_area

↔

- ION
- Pipe
- Socket
- GPUs(kgsl)

...

Requirements for page releasing:

- Synchronized releasing(No cpu switching)

↔

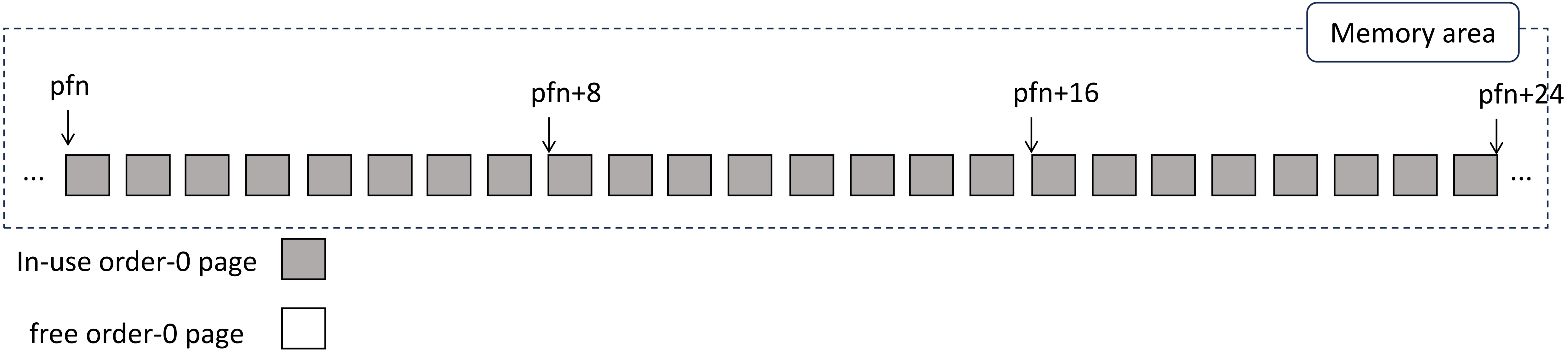
- ION: releasing pages asynchronously
- ***Pipe***
- Socket
- GPUs(kgsl):releasing pages asynchronously

...

# Advancing Towards a More Effective Cross-Cache Attack

Solving Challenge2: Deterministic heap shaping

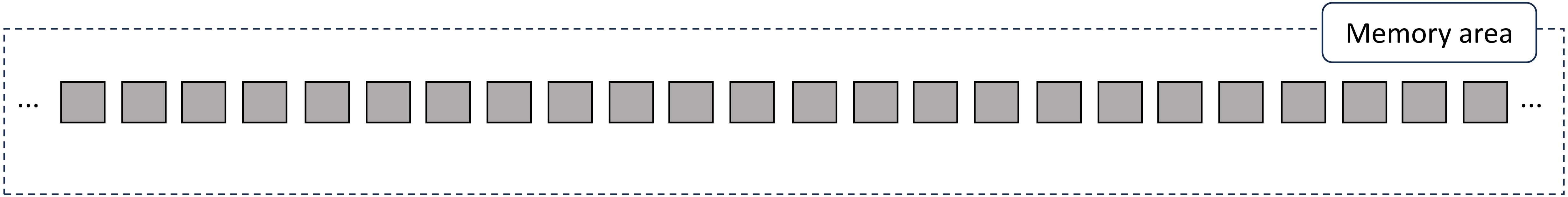
Step3: allocate a few hundreds of ***physically continuous*** order-0 pages from UNMOVALE free_area



# Advancing Towards a More Effective Cross-Cache Attack

Solving Challenge2: Deterministic heap shaping

Step3: allocate a few hundreds of ***physically continuous*** order-0 pages from UNMOVALE free_area




Choosing the proper kernel component:

Requirements for page allocation:
- Able to allocate a large number of order-0 pages
- Allocated from UNMOVALE free_area
- Relatively Clean: No other allocation than allocating order-0 pages

↔

- ION
- Pipe
- Socket
- GPUs(kgsl)

…

Requirements for page releasing:
- Synchronized releasing
- Able to release pages partially

↔

- ION: releasing pages asynchronously
- ***Pipe***
- Socket
- GPUs(kgsl):releasing pages asynchronously

…

# Advancing Towards a More Effective Cross-Cache Attack

Solving Challenge2: Deterministic heap shaping

Step3: allocate a few hundreds of ***physically continuous*** order-0 pages from UNMOVALE free_area

Page allocation and releasing with pipe:

Allocating order-0 page when writing pipe:

```
pipe_write(struct kiocb *iocb, struct iov_iter *from)
{
...
if (bufs < pipe->buffers) {
        int newbuf = (pipe->curbuf + bufs) & (pipe->buffers-1);
        struct pipe_buffer *buf = pipe->bufs + newbuf;
        struct page *page = pipe->tmp_page;
        int copied;

        if (!page) {
           page = alloc_page(GFP_HIGHUSER | __GFP_ACCOUNT);
           if (unlikely(!page)) {
              ret = ret ? : -ENOMEM;
              break;
           }
           pipe->tmp_page = page;
        }
....
```

Releasing order-0 page when reading pipe:

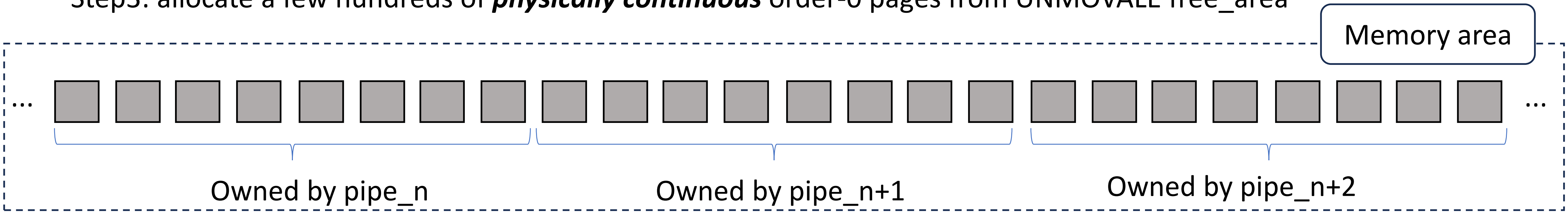
```
static void anon_pipe_buf_release(struct pipe_inode_info *pipe,
              struct pipe_buffer *buf)
{
   struct page *page = buf->page;


   /*
    * If nobody else uses this page, and we don't already have a
    * temporary page, let's keep track of it as a one-deep
    * allocation cache. (Otherwise just release our reference to it)
    */
   if (page_count(page) == 1 && !pipe->tmp_page)
      pipe->tmp_page = page;
   else
      put_page(page);
}
```

(The very first page won't be released, so we need to pre-allocated it before the heap shaping)

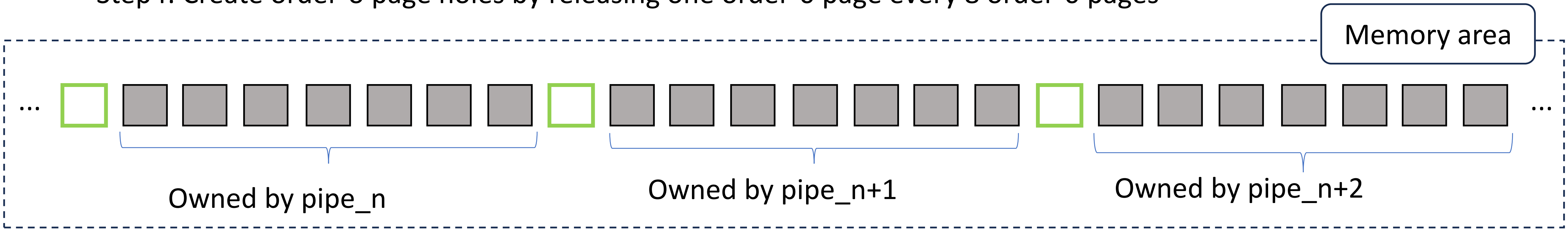
# Advancing Towards a More Effective Cross-Cache Attack

Solving Challenge2: Deterministic heap shaping

Step3: allocate a few hundreds of ***physically continuous*** order-0 pages from UNMOVALE free_area




Step4: Create order-o page holes by releasing one order-0 page every 8 order-0 pages

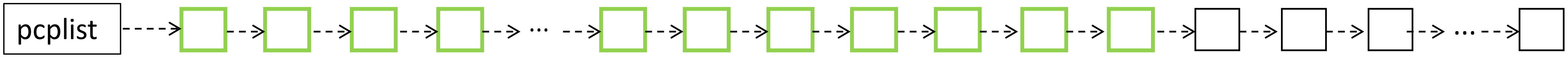



order-0 page hole

# Advancing Towards a More Effective Cross-Cache Attack

Solving Challenge2: Deterministic heap shaping

Pcplist of cpu#0 would be like:




order-0 page hole 

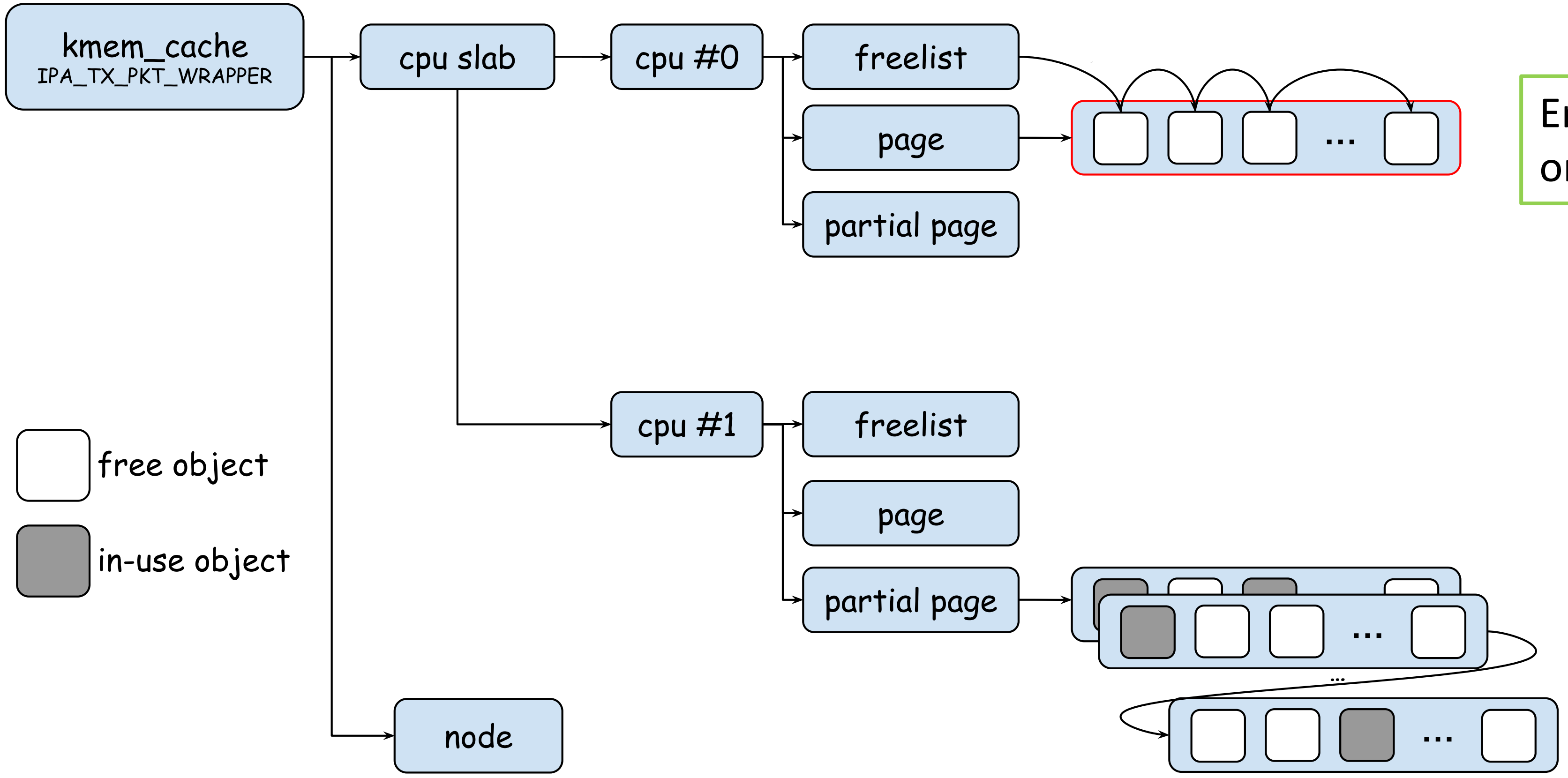
# Advancing Towards a More Effective Cross-Cache Attack

Solving Challenge2: Deterministic heap shaping

Step5. Trigger the step1 in “new optimized workflow of cross cache attack for the issue”

## The optimized workflow of cross cache attack for the issue:

Step1. Defragmentation with race style slab move primitive, a **new** slab will be created:

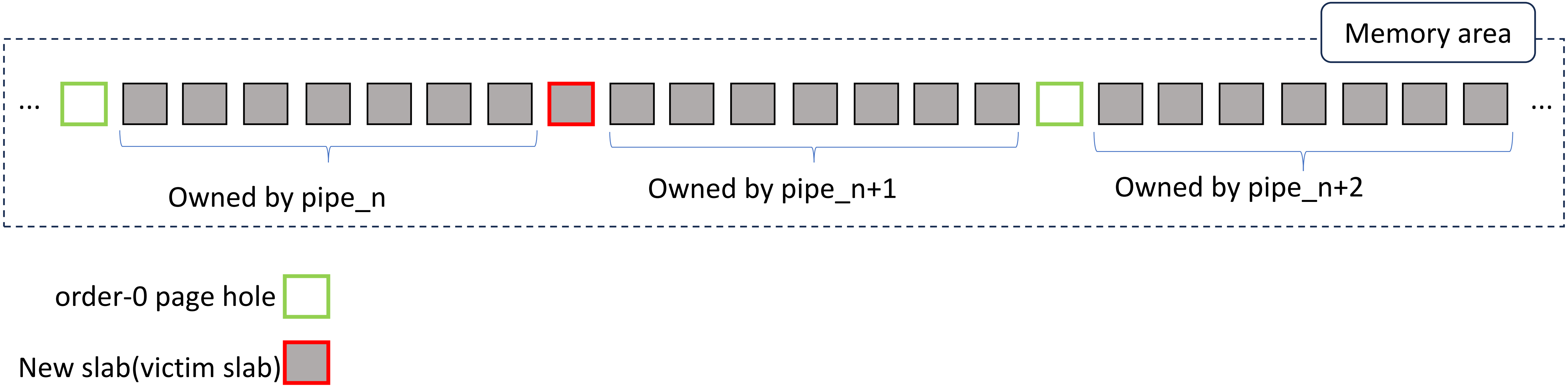



Empty slab comes from order-0 page holes

# Advancing Towards a More Effective Cross-Cache Attack

Solving Challenge2: Deterministic heap shaping

Step5. Trigger the step1 in “new optimized workflow of cross cache attack for the issue”



# Advancing Towards a More Effective Cross-Cache Attack

Solving Challenge2: Deterministic heap shaping

Step6. Occupy all the other order-0 page holes, except the one has been used as new slab

Choosing the proper kernel component:

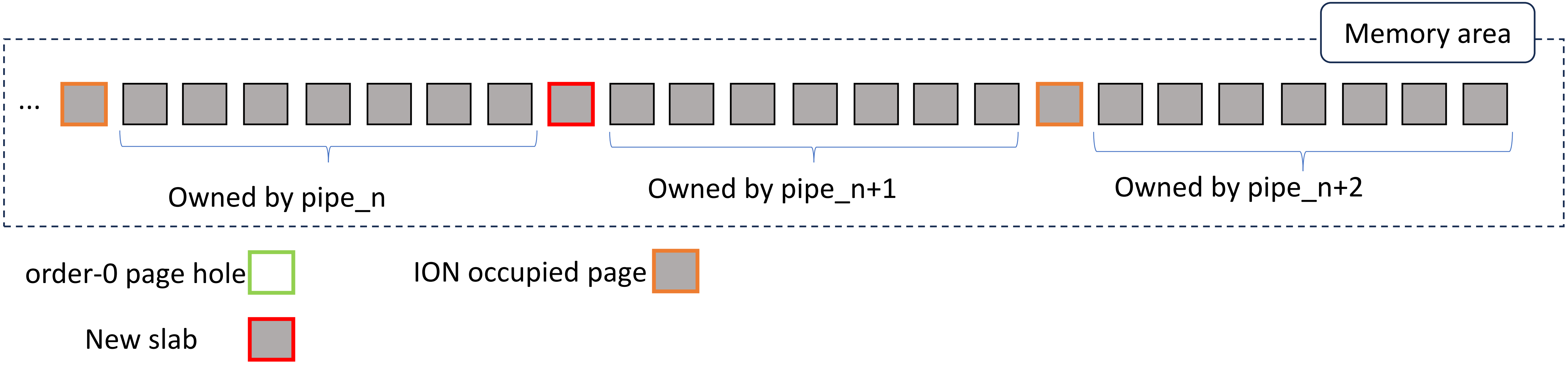
Requirements for page allocation:

- Able to allocate a large number of order-0 pages
- Allocated from UNMOVALE free_area

⟷

- **ION**
- Pipe
- Socket
- GPUs(kgsl)

...

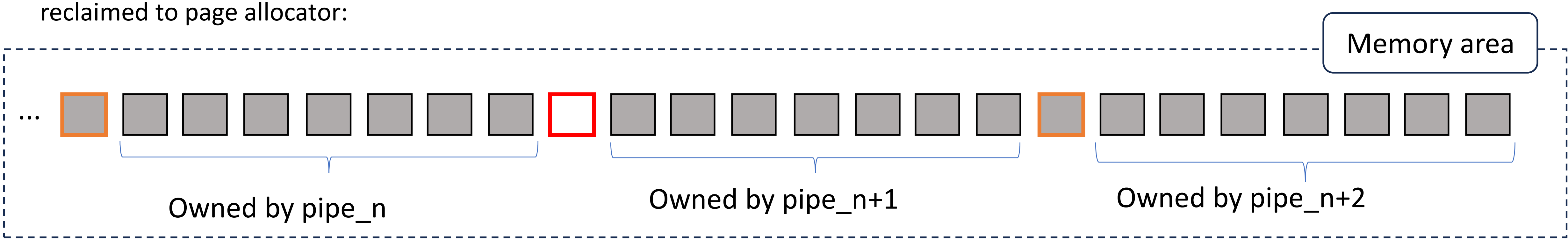


# Advancing Towards a More Effective Cross-Cache Attack

## Solving Challenge2: Deterministic heap shaping

Step7. Finish the step2 ~ step5 of “new optimized workflow of cross cache attack for the issue”

After the step5 of "optimized workflow of cross cache attack for the issue", the victim slab will be reclaimed to page allocator:

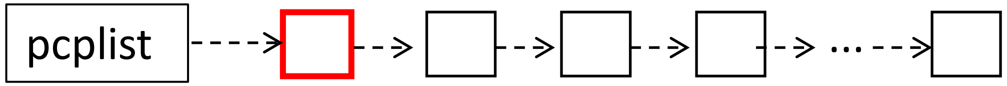



order-0 page hole

ION occupied page

released victim slab

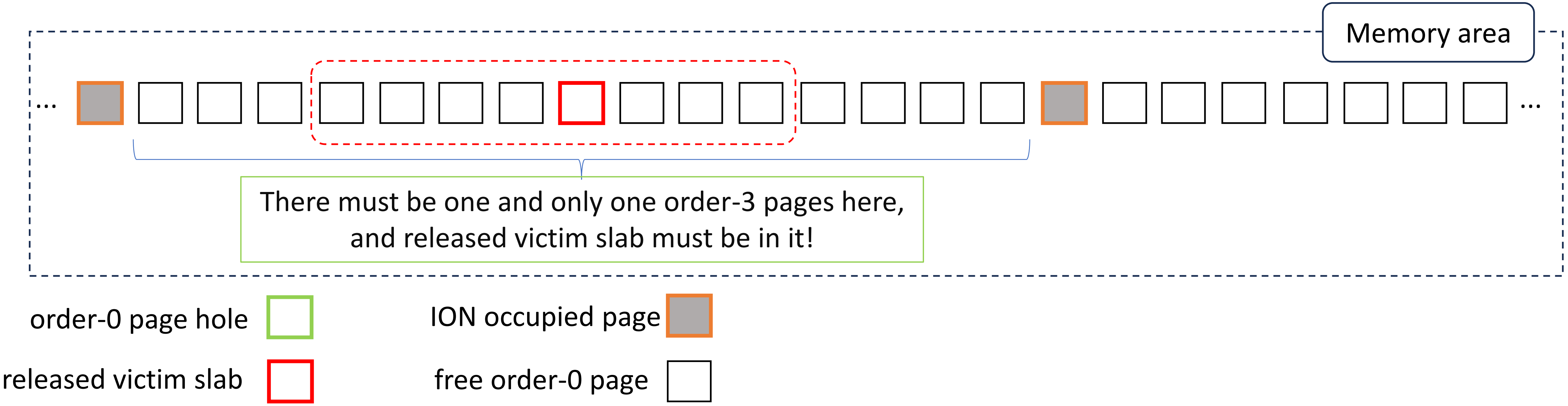
Pcplist of cpu#0 would be like:



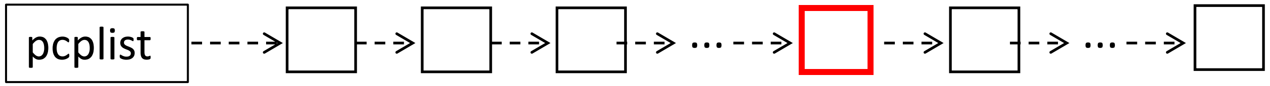
# Advancing Towards a More Effective Cross-Cache Attack

Solving Challenge2: Deterministic heap shaping

Step8. Release all the pages owned by the pipe




Pcplist of cpu#0 would be like:

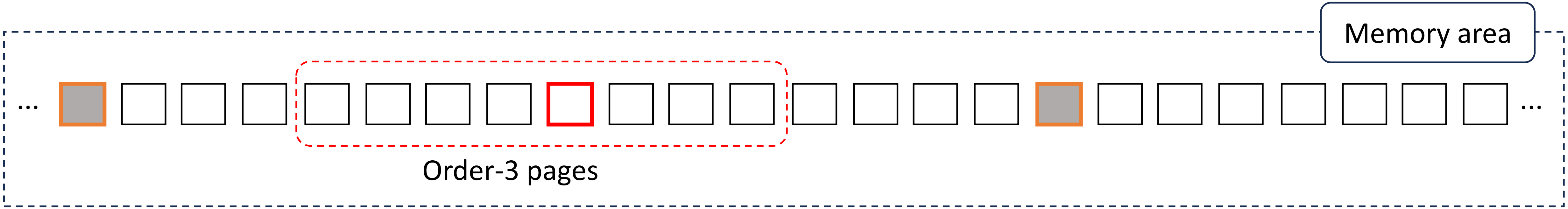


# Advancing Towards a More Effective Cross-Cache Attack

Solving Challenge2: Deterministic heap shaping

Step9. Release all the pages created in step2 to forse the flushing of pcplist

Victim slab and other order-0 pages are reclaimed into free_area, page merging will happen because of "Deterministic page merging"




Step10. Heap spray lots of file array to occupy the order-3 pages where victim slab lies

# Advancing Towards a More Effective Cross-Cache Attack

Solving Challenge2: Deterministic heap shaping

In actual practice, the success rate of the entire utilization largely depends on step 3:

How?

Step3: allocate a few hundreds of ***physically continuous*** order-0 pages from UNMOVALE free_area

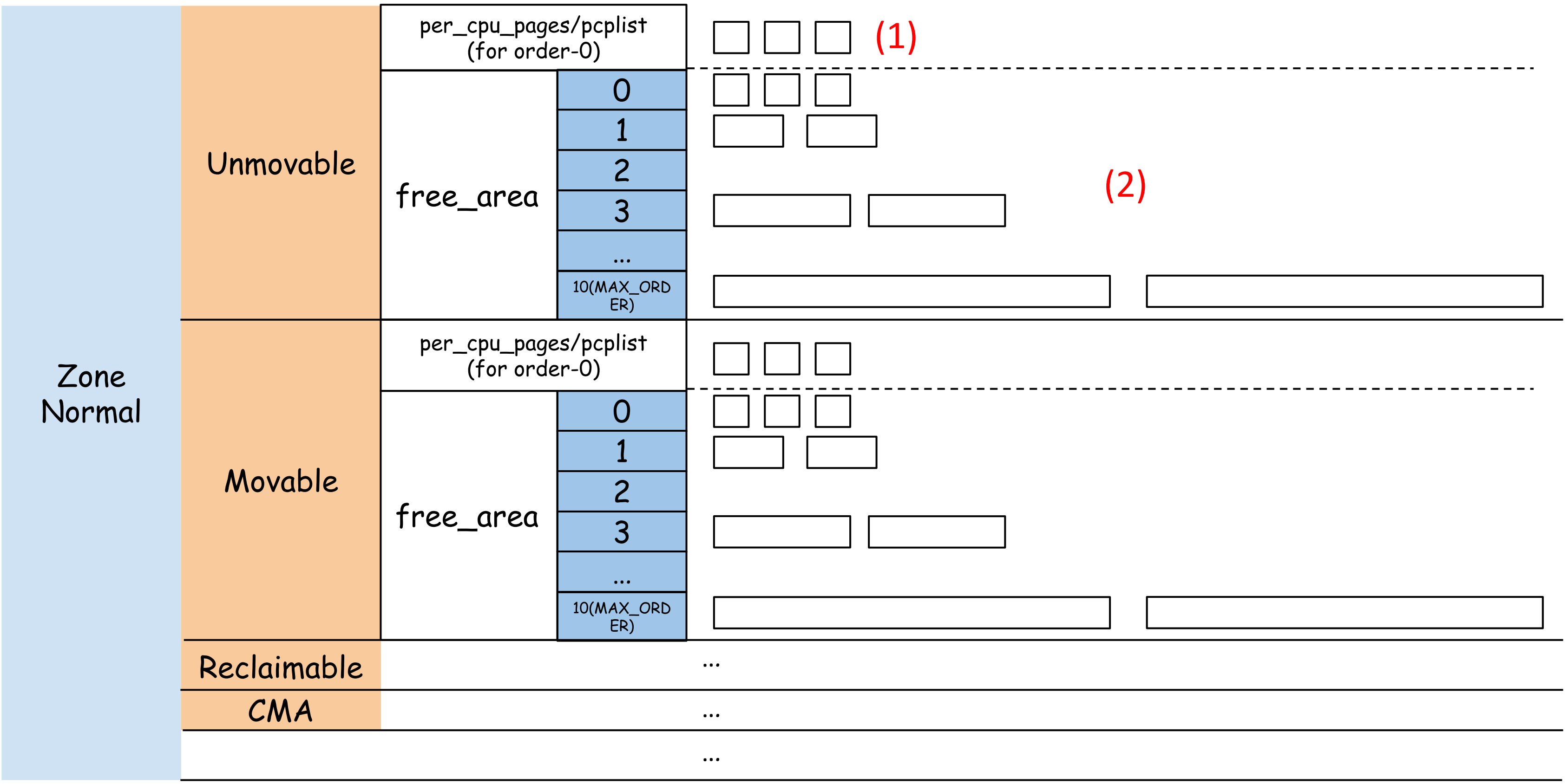
# Advancing Towards a More Effective Cross-Cache Attack

Detect status of page allocator in a side-channel way

If we keeps on allocate order-0 pages  with  "__GFP_KSWAPD_RECLAIM" flag enabled from UNMOVALBE free_area:

State 1:allocated from pcplist first

State 2:pcplist become empty, Unmovable free_area will be used:
Start from low-order



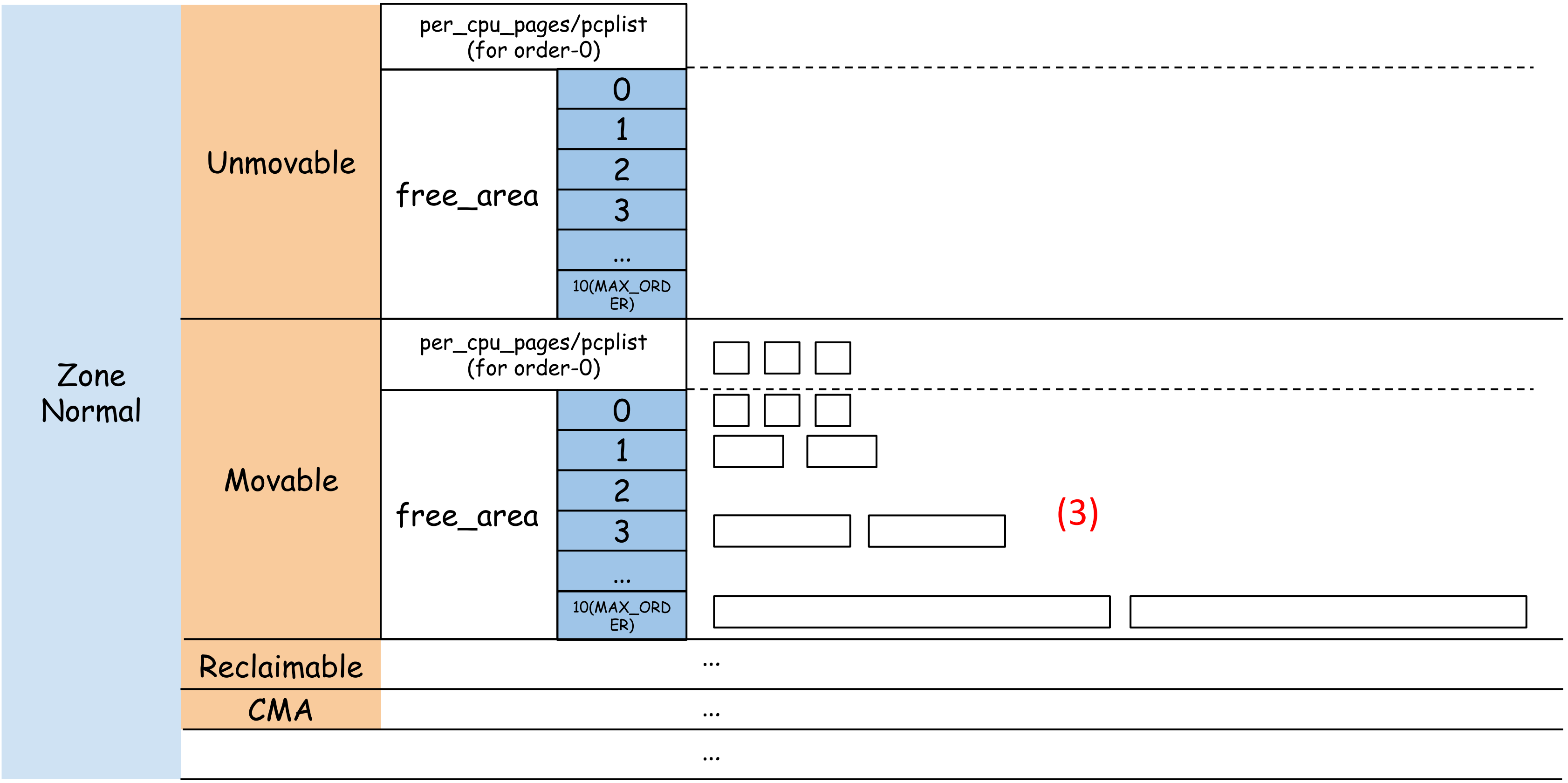
# Advancing Towards a More Effective Cross-Cache Attack

Detect status of page allocator in a side-channel way

If we keeps on allocate order-0 pages with "__GFP_KSWAPD_RECLAIM" flag enabled from UNMOVALBE free_area:

State3: If Unmovable free_area becom empty, other migration type free_areas will be used for allocation acording to fallback list

Wake up kswapd for reclaiming pages if free pages of zone is under High watermark.




```
static int fallbacks[MIGRATE_TYPES][4] = {
   [MIGRATE_UNMOVABLE]   = { MIGRATE_RECLAIMABLE, MIGRATE_MOVABLE,   MIGRATE_TYPES },
......
};
```

# Advancing Towards a More Effective Cross-Cache Attack

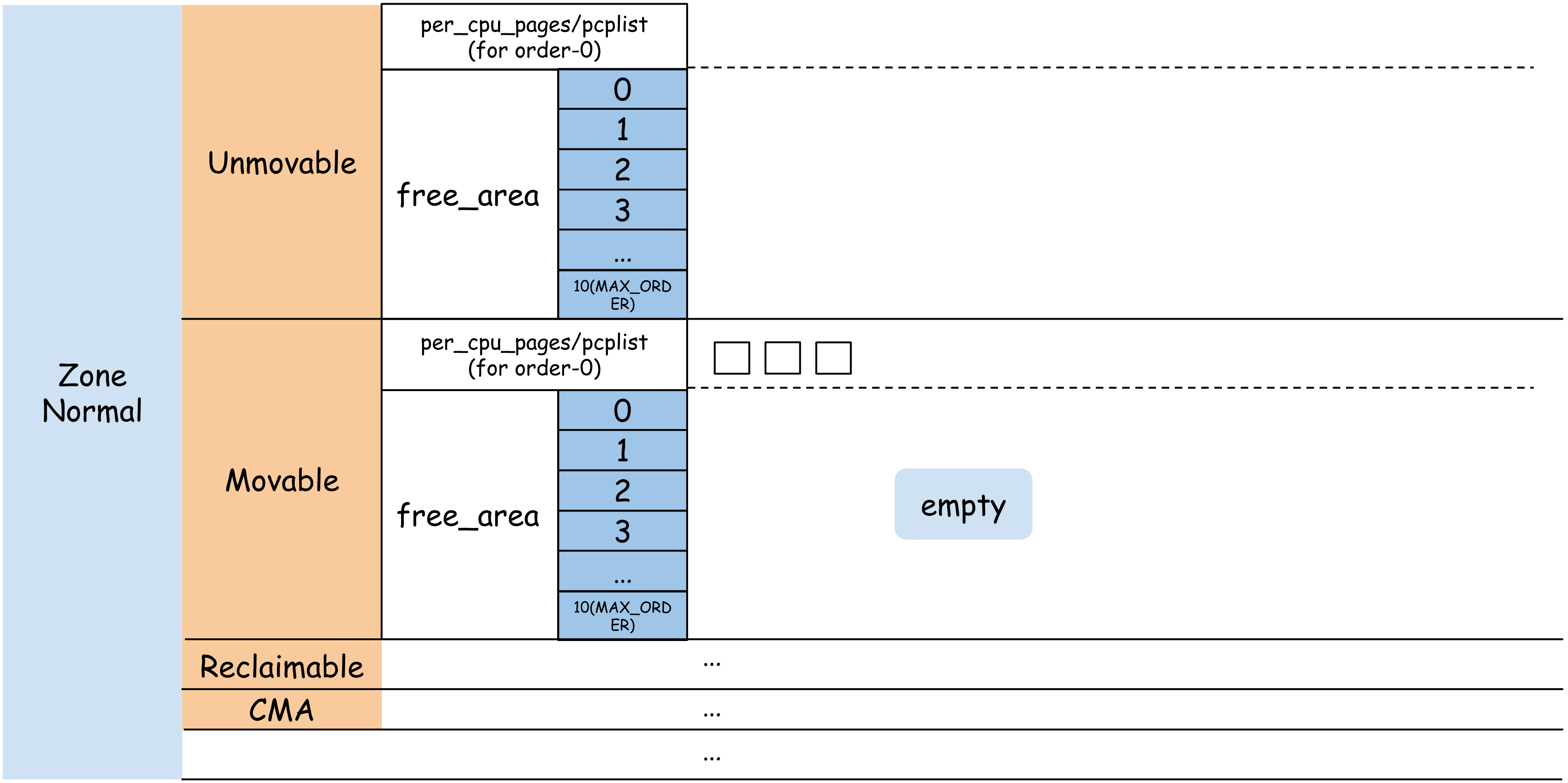
Detect status of page allocator in a side-channel way

If we keeps on allocate order-0 pages with "__GFP_KSWAPD_RECLAIM" flag enabled from UNMOVALBE free_area:

State 4: If other migration type free_areas becom empty, then enter the slow path for allocating order-0 page:

- Wake up kswpad for reclaiming pages
- Direct reclaim

...



# Advancing Towards a More Effective Cross-Cache Attack

Detect status of page allocator in a side-channel way

If we keeps on allocate order-0 pages with "__GFP_KSWAPD_RECLAIM" flag enabled from UNMOVALBE free_area:

Reclaming pages:

- Wake up kswpad for reclaiming pages
- direct reclaim

←

- LRU_INACTIVE_ANON
- LRU_INACTIVE_FILE
- LRU_ACTIVE_ANON
- LRU_ACTIVE_FILE
- shrinker_list

# Advancing Towards a More Effective Cross-Cache Attack

Detect status of page allocator in a side-channel way

- LRU_ACTIVE_ANON
- LRU_INACTIVE_ANON
- LRU_ACTIVE_FILE
- LRU_INACTIVE_FILE
- shrinker_list

Exported by /proc/meminfo, accessable from untrusted app:

```
x1q:/ $ cat /proc/meminfo
MemTotal:       10856332 kB
MemFree:         3721612 kB
MemAvailable:    4817668 kB
Buffers:           15240 kB
Cached:          3442100 kB
SwapCached:        66132 kB
Active:          2993296 kB
Inactive:        2225988 kB
Active(anon):    1326992 kB
Inactive(anon):   451772 kB
Active(file):    1666304 kB
Inactive(file):  1774216 kB
Unevictable:        3652 kB
Mlocked:            3652 kB
RbinTotal:             0 kB
RbinAlloced:           0 kB
RbinPool:              0 kB
RbinFree:              0 kB
RbinCached:            0 kB
ZeroedFree:        35516 kB
SwapTotal:       4194300 kB
SwapFree:        3806356 kB
Dirty:                52 kB
Writeback:             0 kB
AnonPages:       1737988 kB
Mapped:          1203732 kB
Shmem:             14180 kB
KReclaimable:     333184 kB
```

# Advancing Towards a More Effective Cross-Cache Attack

Detect status of page allocator in a side-channel way

```
x1q:/ $ cat /proc/meminfo
MemTotal:       10856332 kB
MemFree:         3721612 kB
MemAvailable:    4817668 kB
Buffers:           15240 kB
Cached:          3442100 kB
SwapCached:        66132 kB
Active:          2993296 kB
Inactive:        2225988 kB
Active(anon):    1326992 kB
Inactive(anon):   451772 kB
Active(file):    1666304 kB
Inactive(file):  1774216 kB
Unevictable:        3652 kB
Mlocked:            3652 kB
RbinTotal:             0 kB
RbinAlloced:           0 kB
RbinPool:              0 kB
RbinFree:              0 kB
RbinCached:            0 kB
ZeroedFree:        35516 kB
SwapTotal:       4194300 kB
SwapFree:        3806356 kB
Dirty:                52 kB
Writeback:             0 kB
AnonPages:       1737988 kB
Mapped:          1203732 kB
Shmem:             14180 kB
KReclaimable:     333184 kB
```

Get reduced frequently

Page allocator might be in State 3 or State 4

Unmovable free_area is almost empty!

# Advancing Towards a More Effective Cross-Cache Attack

Detect status of page allocator in a side-channel way

Tested on the device with kernel 4.14:

/proc/pagetypeinfo:

```
[+] value for evaluating the reclaiming:21
[+] value for evaluating the reclaiming:21
[+] value for evaluating the reclaiming:22
[+] value for evaluating the reclaiming:23
[+] value for evaluating the reclaiming:24
[+] value for evaluating the reclaiming:24
[+] value for evaluating the reclaiming:25
pagetypeinfo:
Page block order: 10
Pages per block:  1024

Free pages count per migrate type at order       0      1      2      3      4      5      6      7      8      9     10
Node    0, zone   Normal, type    Unmovable      0      0      6      7      2      2      0      0      0      0      0
Node    0, zone   Normal, type      Movable  37767   2593    750    327    129     32      0      0      0      0      0
Node    0, zone   Normal, type  Reclaimable    115    107    249    144     29      4      0      0      0      0      0
Node    0, zone   Normal, type          CMA    589    127     11      1      2      0      0      0      0      0      0
Node    0, zone   Normal, type   HighAtomic      0      2      4      5      5      2      3      2      1      0      0
Node    0, zone   Normal, type      Isolate      0      0      0      0      0      0      0      0      0      0      0

Number of blocks type     Unmovable      Movable  Reclaimable          CMA   HighAtomic      Isolate
Node 0, zone    Normal         1253         1589           33          112            1            0
```

# Advancing Towards a More Effective Cross-Cache Attack

Strategy for allocating a few hundreds of ***physically continuous*** order-0 pages from UNMOVALE free_area:

Step1: reserve a dozen of order-8/9 pages with ION

```
#if defined(CONFIG_IOMMU_IO_PGTABLE_ARMV7S)
static const unsigned int orders[] = {8, 4, 0};
#else
static const unsigned int orders[] = {9, 4, 0};
#endif
```

Step2: Create and detect the empty state of Unmovable free_area:

2.1: Consume a large memory from both Unmoable free_area and Movable free_area. This will put memory of zone under pressure(for example, under High watermark )

```
Allocate_large_memory _with_ION(); // Consume a large memory from both Unmoable free_area
Allocate_large_memory_with_mmap(); // Consume a large memory from both Moable free_area
```

2.2: Run the circle to detect the empty state of Unmovable free_area

```
While (1) {
        Allocate_a_few_order0_pages();
        Detect_page_allocator_state_by_watching_meminfo();
        If (page_allocator_enter_state_3_or_4) {
                break;
        }
}
```

# Advancing Towards a More Effective Cross-Cache Attack

Strategy for allocating a few hundreds of ***physically continuous*** order-0 pages from UNMOVALE free_area:

Step3: release the order-8 pages with ION

```
pagetypeinfo:
Page block order: 10
Pages per block:  1024

Free pages count per migrate type at order       0      1      2      3      4      5      6      7      8      9     10
Node    0, zone   Normal, type    Unmovable      2      3      7      8      2      2      0      0      0     42     79
Node    0, zone   Normal, type      Movable  38028   2616    757    334    129     33      0      0      0      0      0
Node    0, zone   Normal, type  Reclaimable    115    107    249    144     29      4      0      0      0      0      0
Node    0, zone   Normal, type          CMA    587    127     11      1      2      0      0      0      0      0      0
Node    0, zone   Normal, type   HighAtomic      0      2      4      5      5      2      3      2      1      0      0
Node    0, zone   Normal, type      Isolate      0      0      0      0      0      0      0      0      0      0      0

Number of blocks type     Unmovable      Movable  Reclaimable          CMA   HighAtomic      Isolate
Node 0, zone   Normal          1253         1589           33          112            1            0
```

Step4: allocate some order-0 pages to reduce the noise

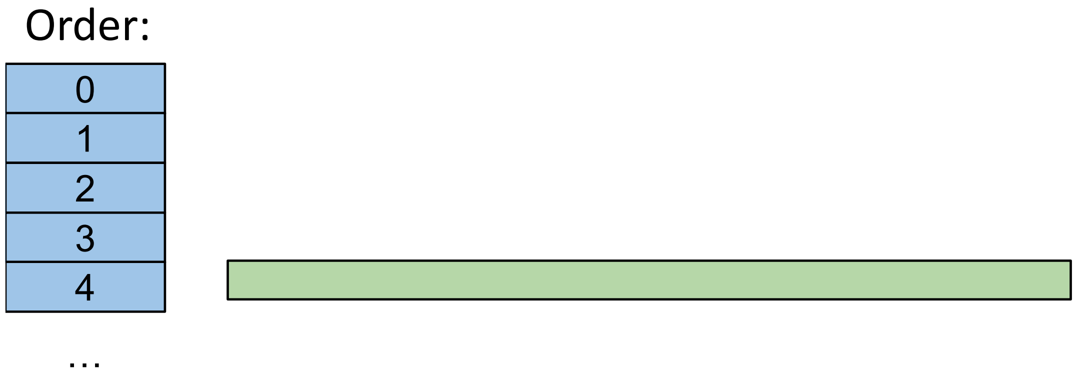
Step5: allocate a few hundreds of ***physically continuous*** order-0 pages from UNMOVALE free_area

# Advancing Towards a More Effective Cross-Cache Attack

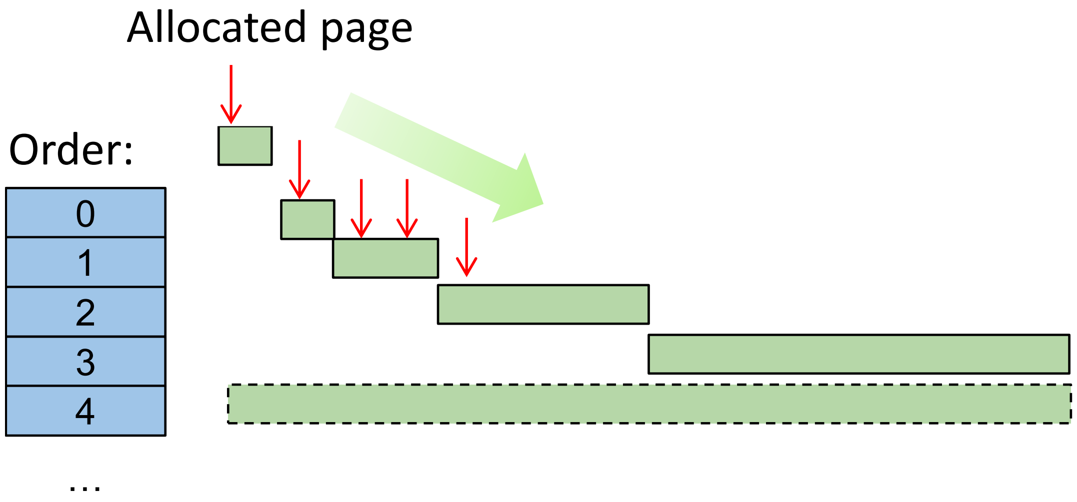
Strategy for allocating a few hundreds of ***physically continuous*** order-0 pages from UNMOVALE free_area:

Step5: allocate a few hundreds of order-0 pages from UNMOVALE free_area

The order-0 page comes from the spliting of high-order pages:




Original state of Unmovable free_area

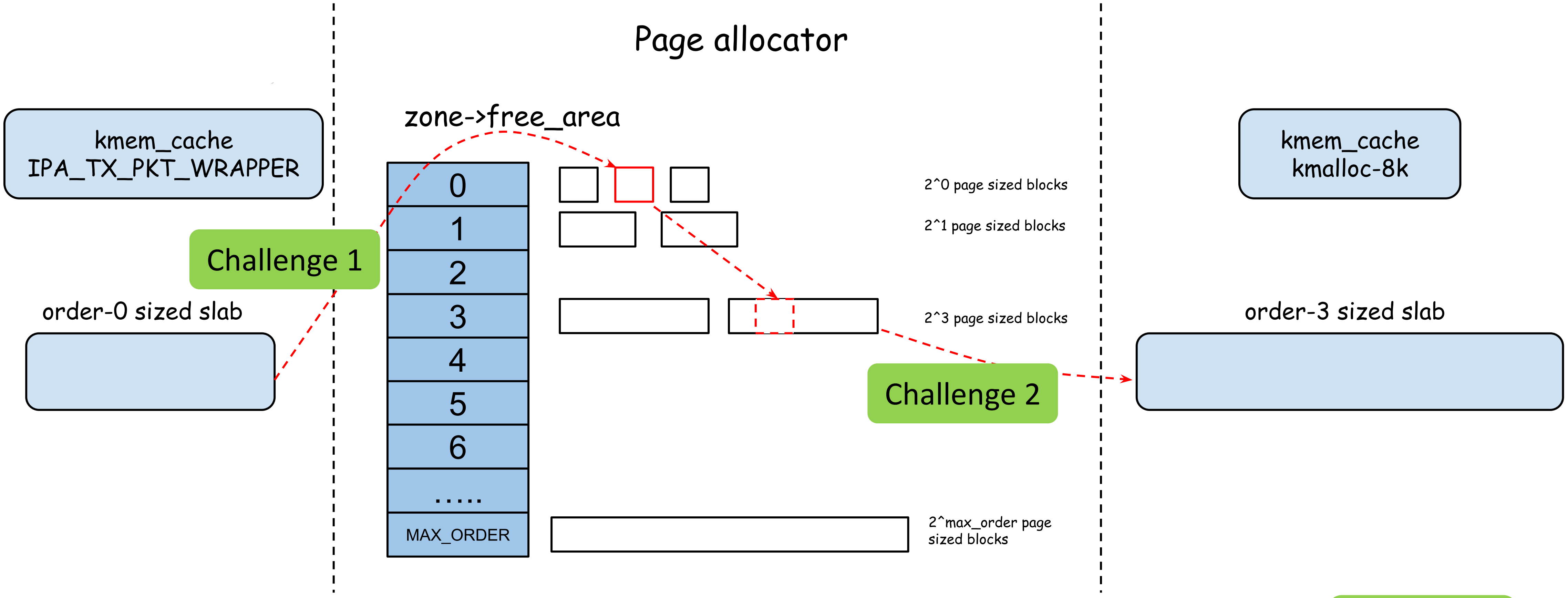



Allocate one order-0 page

So these order-0 pages will be ***physically continuous***

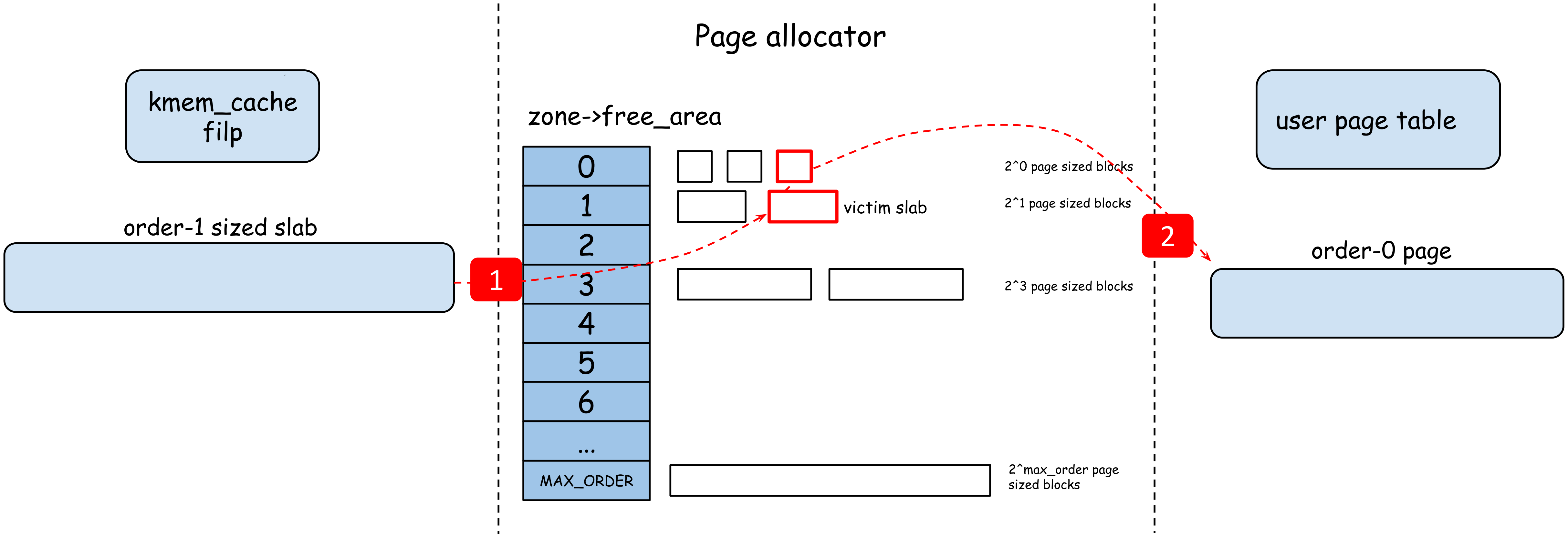
# Advancing Towards a More Effective Cross-Cache Attack

- Challenge 2: How to make order-3 slab reuse the order-0 slab deterministically




- Challenge 1: How to discard the victim order-0 slab under a constrained allocation primitive **SOLVED!**
- Challenge 2: How to make order-3 slab reuse the order-0 slab deterministically **SOLVED!**

# Exploit File UAF with Dirty Pagetable




1: Use the old method to discard the victim filp slab

2: Occupy the released victim filp slab with user page table by heap spraying many user page tables

# Exploit File UAF with Dirty Pagetable

Step1. Use the mentioned method to make Unmovable free_area become almost empty

Step2. Discard the victim filp slab

The occupation is more likely to succeed because the free_area is relatively clean.

Step3. Heap spray many user page tables to occupy the released victim filp slab.

# Exploit File UAF with Dirty Pagetable

Adapt Dirty Pagetable to Samsung Device

Mitigations on Samsung Device:

- Physical KASLR
- RO kernel text

Not working :(

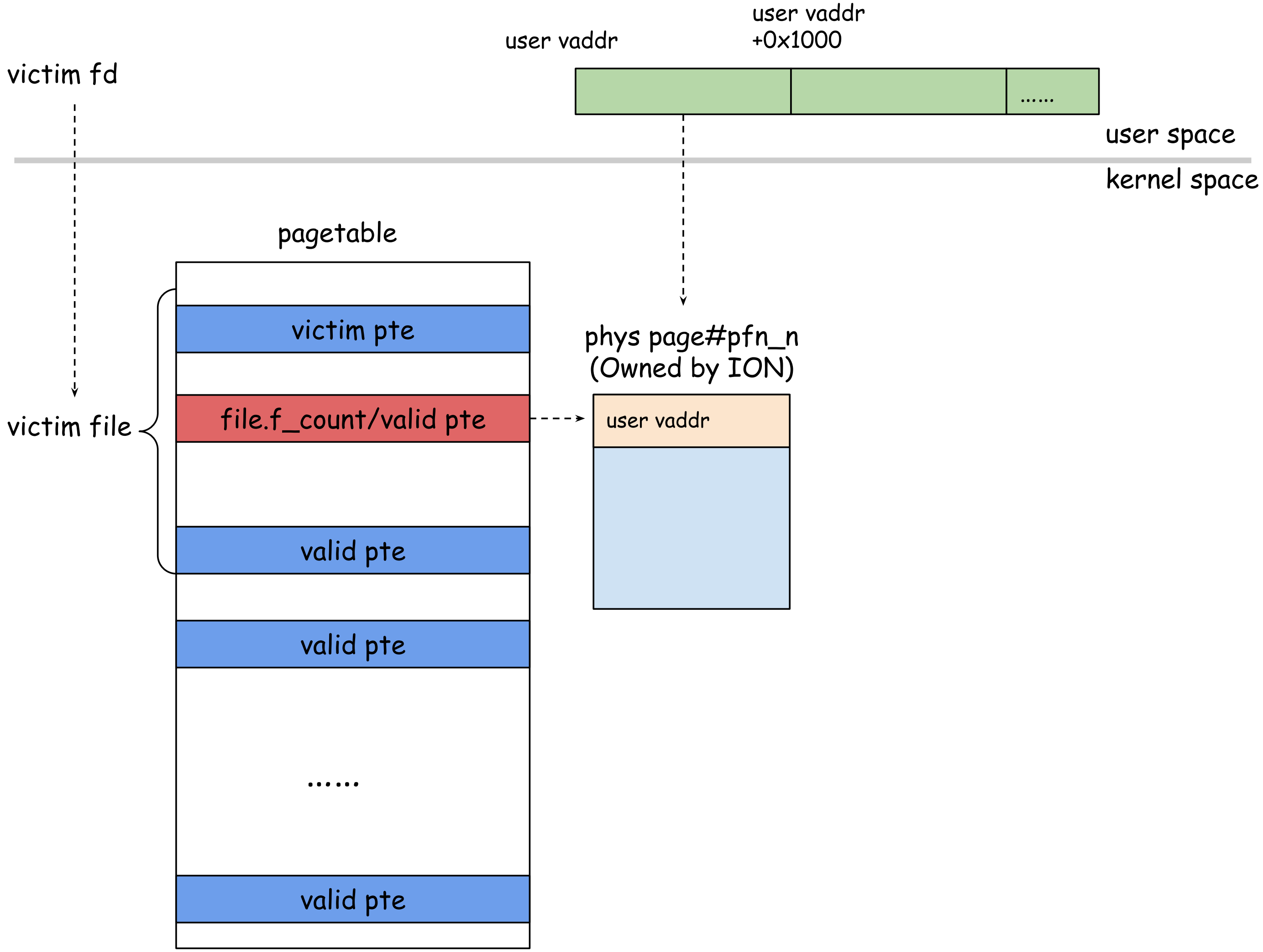
Construct physical AARW with Dirty Pagetable:

https://yanglingxi1993.github.io/dirty_pagetable/dirty_pagetable.html

# Exploit File UAF with Dirty Pagetable

Adapt Dirty Pagetable to Samsung Device

Corrupt kernel object to construct AARW

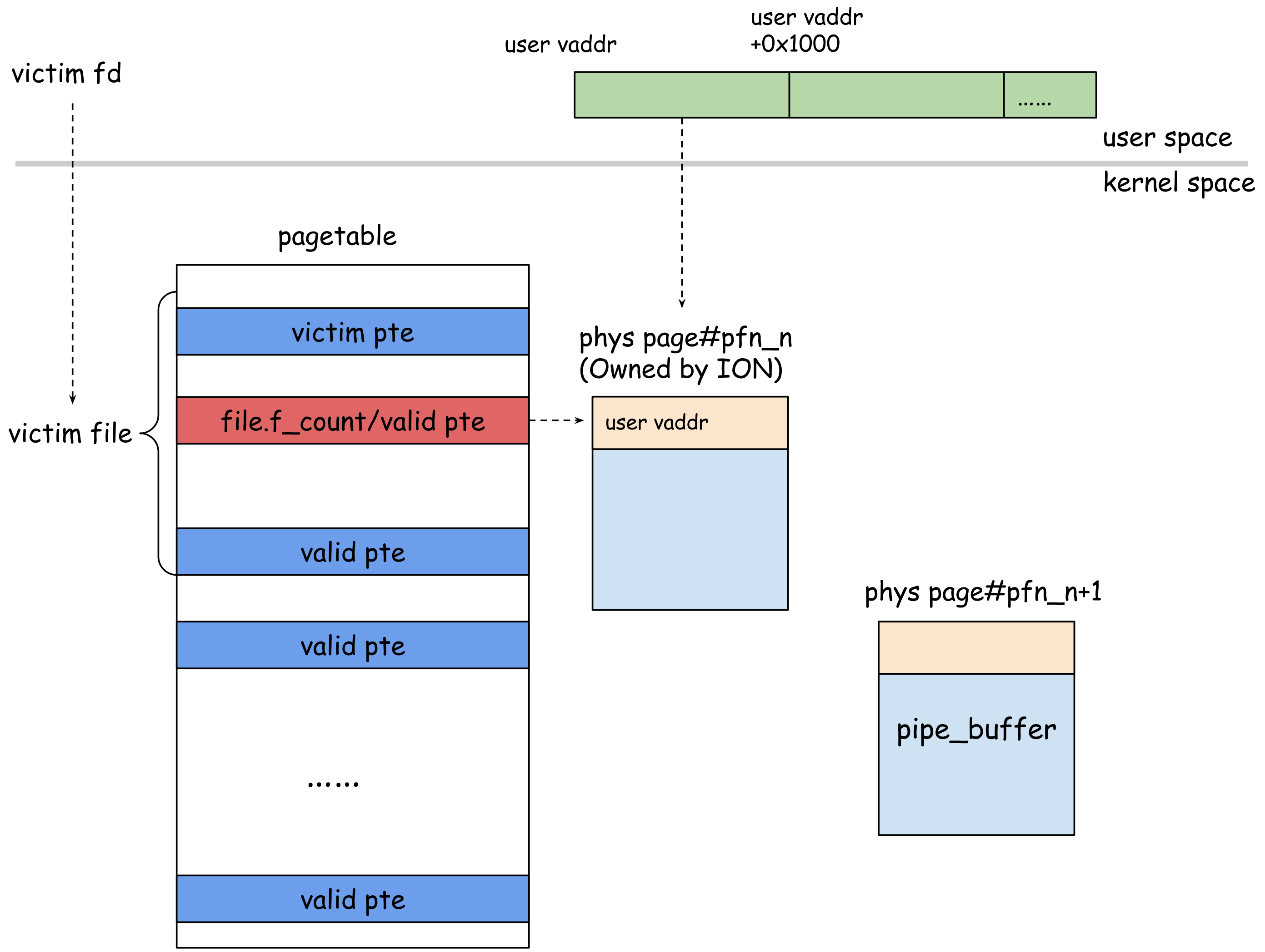


# Exploit File UAF with Dirty Pagetable

Adapt Dirty Pagetable to Samsung Device

Corrupt pipe_buffer to construct AARW

- Make the page of pipe buffer follow the page owned by ION:
  Using the similar technique for allocating physically continuous order-0 pages.



# Exploit File UAF with Dirty Pagetable

Adapt Dirty Pagetable to Samsung Device

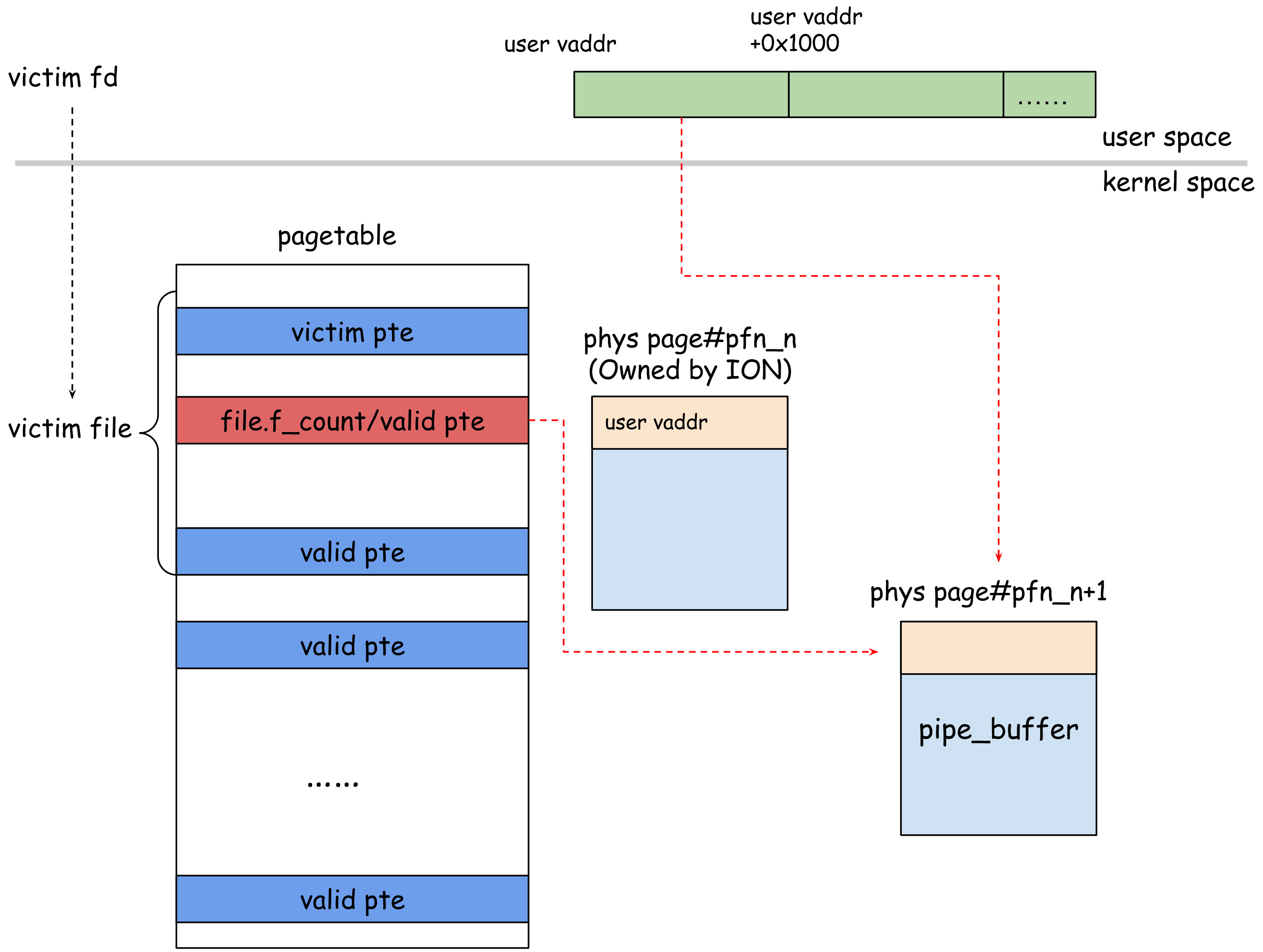
Corrupt kernel object to construct virtual AARW

- Make the page of pipe buffer follow the page owned by ION:
  Using the similar technique for allocating physically continuous order-0 pages.

- victim_pte += 0x1000

```
for(int i = 0; i < 0x1000; i++) {
   dup(victim_fd);
}
```

- Using pipe primitive to construct AARW!



# Bypass SELinux in Samsung device

Attack global data used in "security_compute_av()":

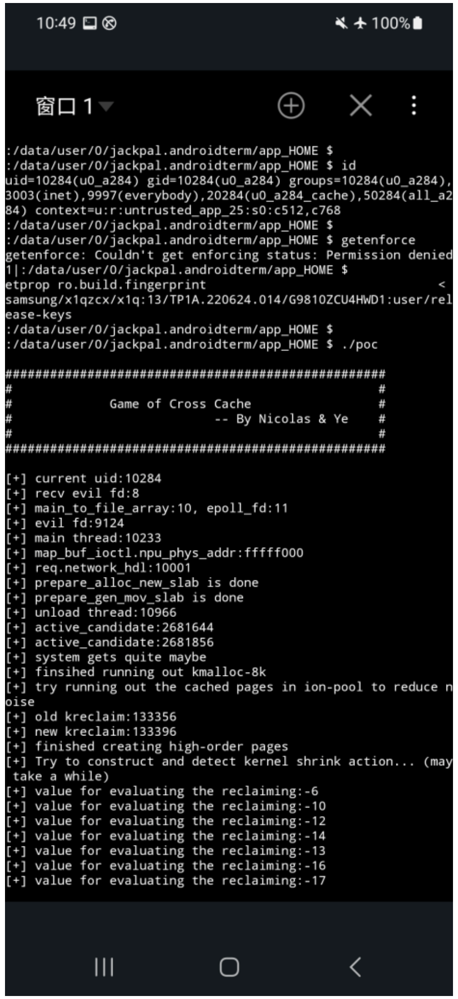
```
void security_compute_av(u32 ssid,
          u32 tsid,
          u16 orig_tclass,
          struct av_decision *avd, ...)
{
   u16 tclass;
   struct context *scontext = NULL, *tcontext = NULL;

   read_lock(&policy_rwlock);
   avd_init(avd);
   xperms->len = 0;
   if (!ss_initialized)
      goto allow;
   ...
   tclass = unmap_class(orig_tclass);
   ...
   context_struct_compute_av(scontext, tcontext, tclass, avd, xperms);
   map_decision(orig_tclass, avd, policydb.allow_unknown);
out:
   read_unlock(&policy_rwlock);
   return;
allow:
   avd->allowed = 0xffffffff;
   goto out;
}
```

```
static void map_decision(u16 tclass, struct av_decision *avd,
          int allow_unknown)
{
   if (tclass < current_mapping_size) {
      unsigned i, n = current_mapping[tclass].num_perms;
      u32 result;

      for (i = 0, result = 0; i < n; i++) {
         if (avd->allowed & current_mapping[tclass].perms[i])
            result |= 1<<i;
         if (allow_unknown && !current_mapping[tclass].perms[i])
            result |= 1<<i;
      }
      avd->allowed = result;

      ...
   }
}
```

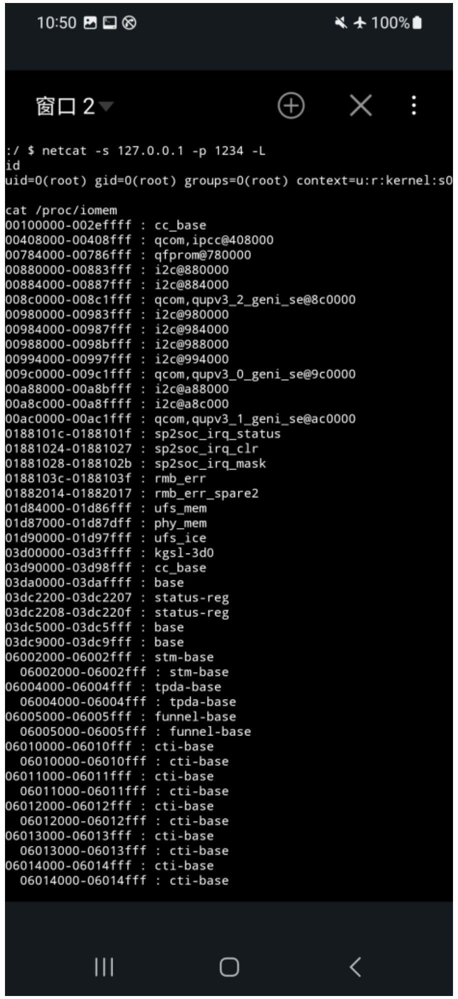
# Win The Game

- System privilege required
- Less than 10% success rate

- Attack from Untrusted App
- ~65%(13/20) success rate







# Mitigations for Cross-cache Attack

SLAB_VIRTUAL:
https://github.com/thejh/linux/commit/bc52f973a53d0b525892088dfbd251bc934e3ac3

Kill the Game!

# Summary

- Advancing Towards a More Effective Cross-Cache Attack
  - Solve the challenge 1: Discard the victim order-0 slab under a really limitation allocation primitive
  - Solve the challenge 2: How to make order-3 slab reuse the order-0 slab deterministically
- Dirty Pagetable on Samsung Device

# Acknowledgements

Ye Zhang, Teacher Jin

# Q&A