cs5965 Advanced OS Implementation

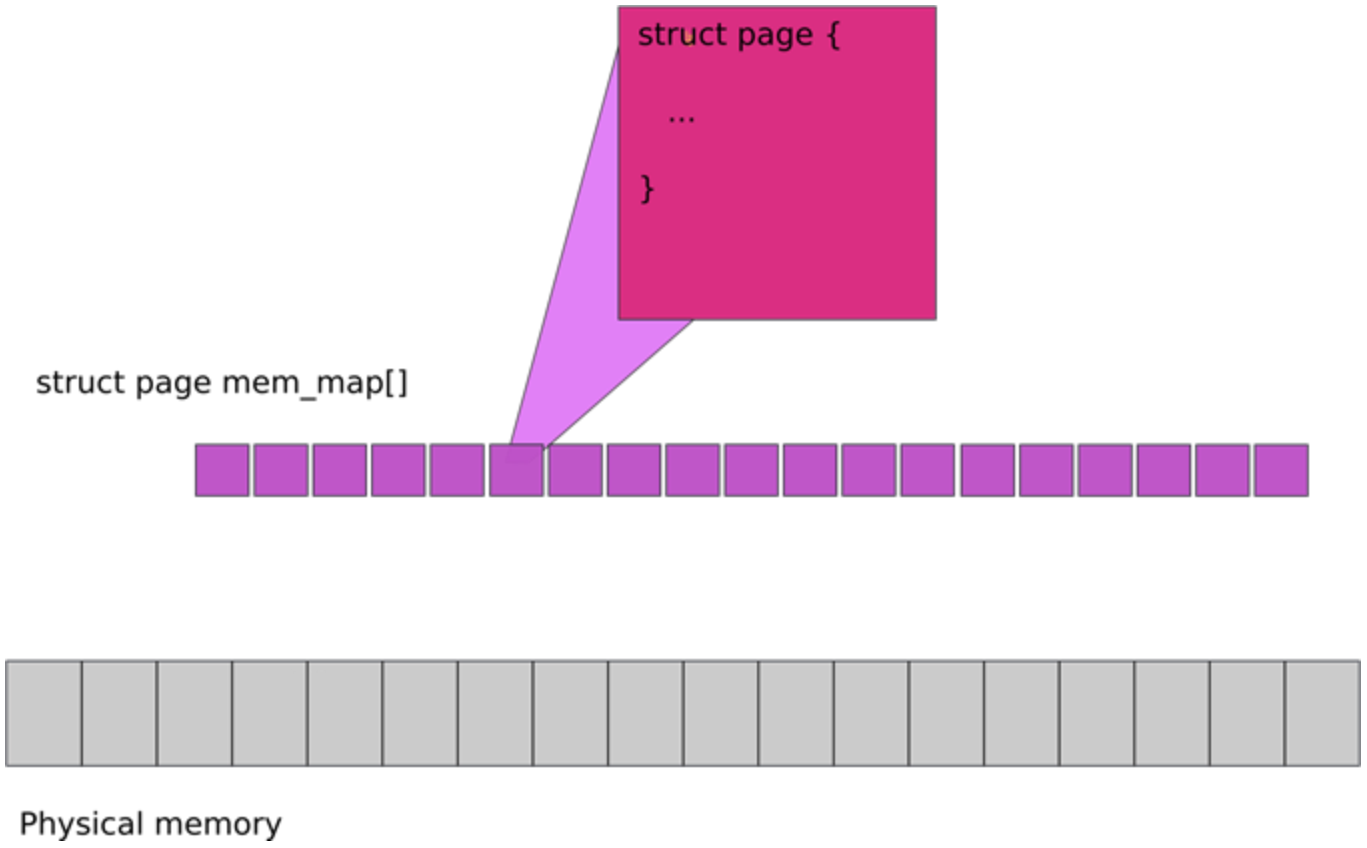
Lecture 05 – Memory management

Anton Burtsev

Physical memory

Physical memory

We need a smaller array to describe physical pages, e.g., mem_map[] in Linux



- Memory allocation

Simplest memory allocator

- Bitmap of all pages
- Bootmem allocator in Linux

- Allocation searches for an unused page
- Multiple sub-page allocations can be served from the same page by advancing a pointer

- Works ok, but what is the problem?

Boot memory allocator

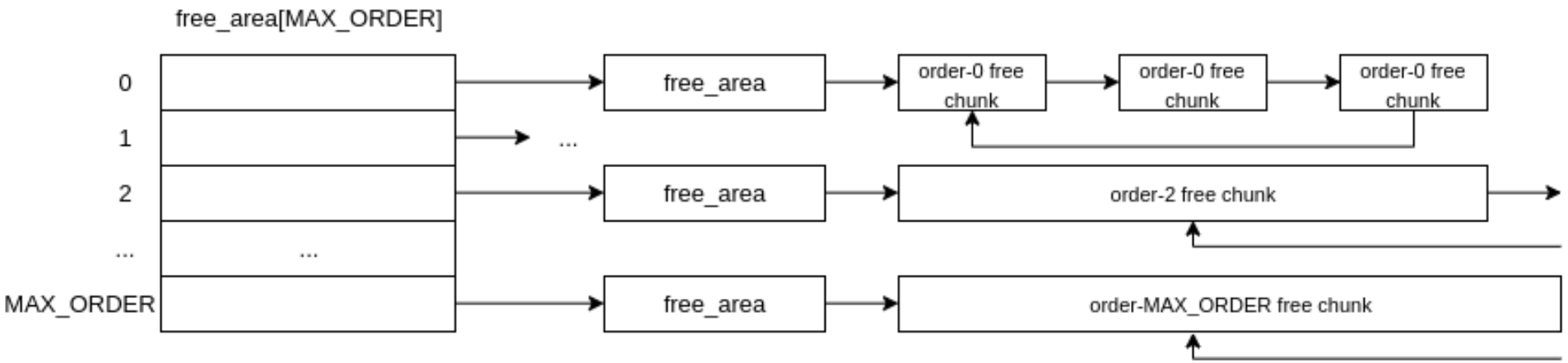
- Bitmap of all pages
- Bootmem allocator in Linux

- Allocation searches for an unused page
- Multiple sub-page allocations can be served from the same page by advancing a pointer

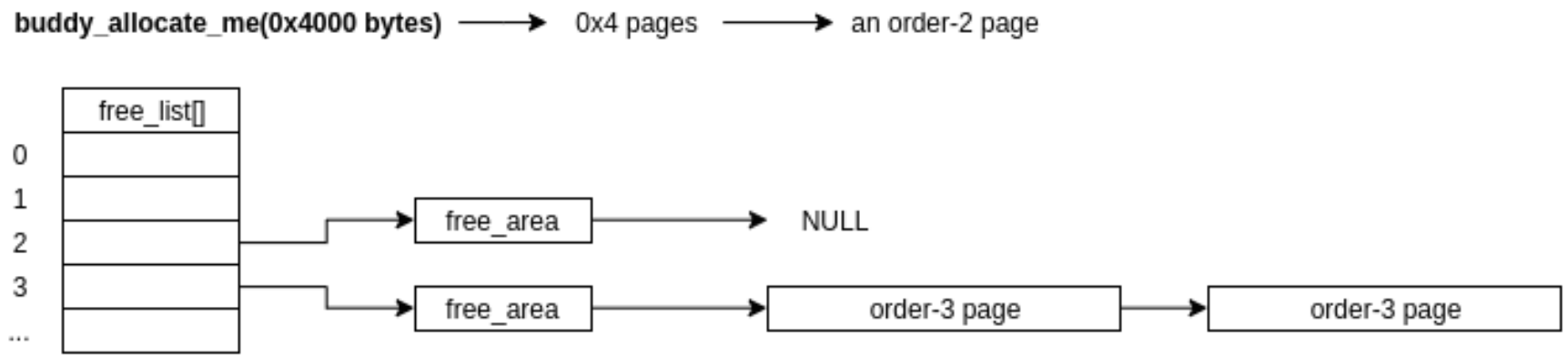
- Works ok, but what is the problem?
- Linear scan of the bitmap
  - Too long

- Buddy:
- Physical Memory Allocator

Buddy memory allocator



Buddy memory allocator



What's wrong with buddy?

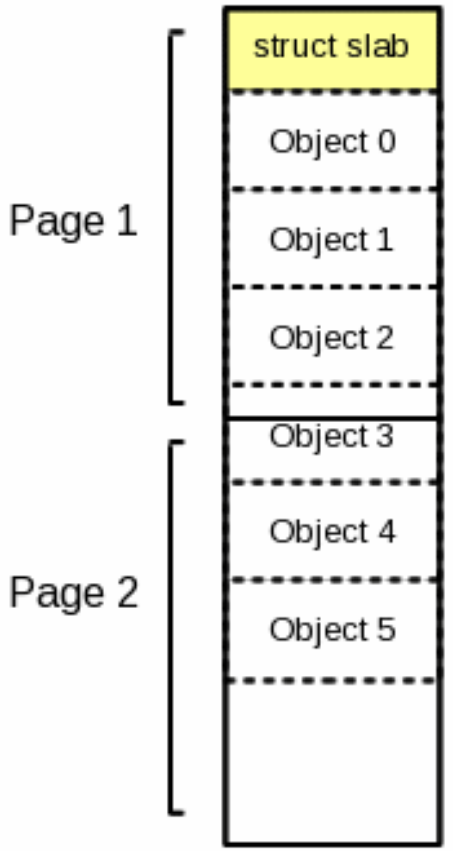
What's wrong with buddy?

- Buddy allocator is ok for large allocations
  - E.g. 1 page or more
- But what about small allocations?
- Buddy uses the whole page for a 4 bytes allocation
  - Wasteful
- Buddy is still slow for short-lived objects

- Slab:
- Allocator for object of a fixed size

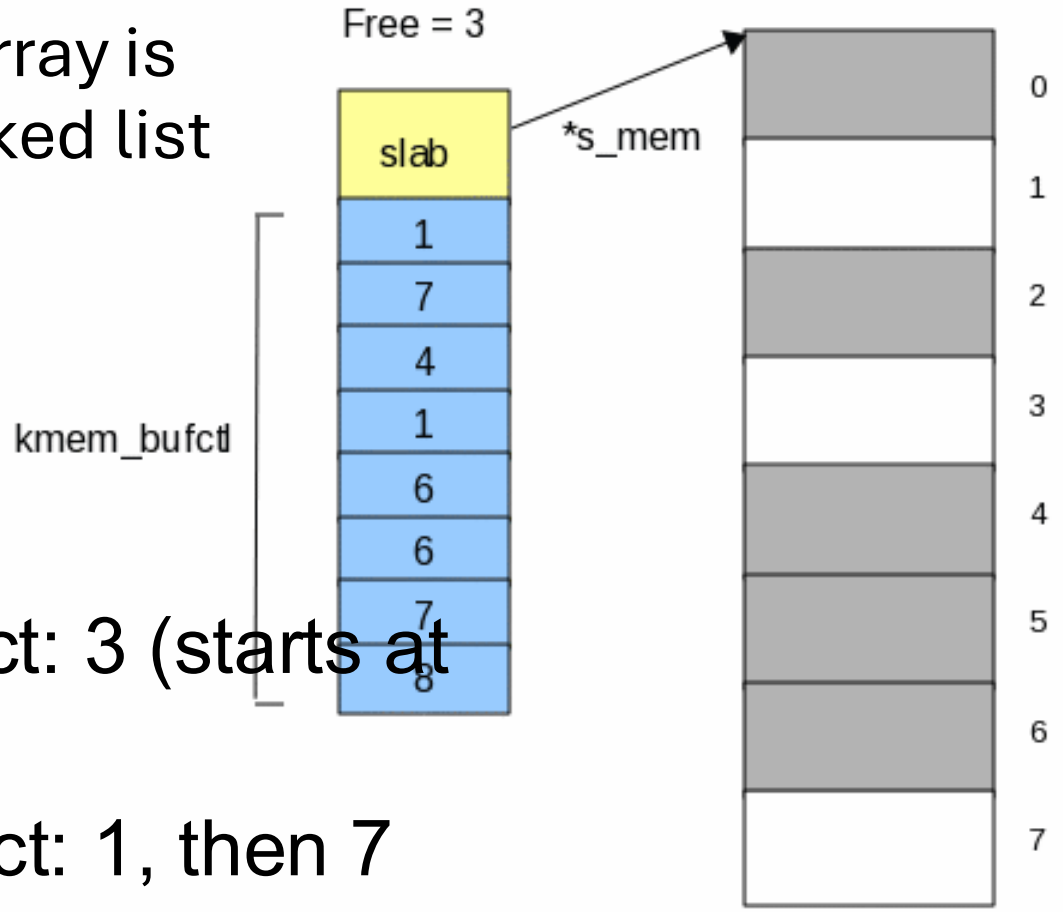
Slab

- A 2 page slab with 6 objects



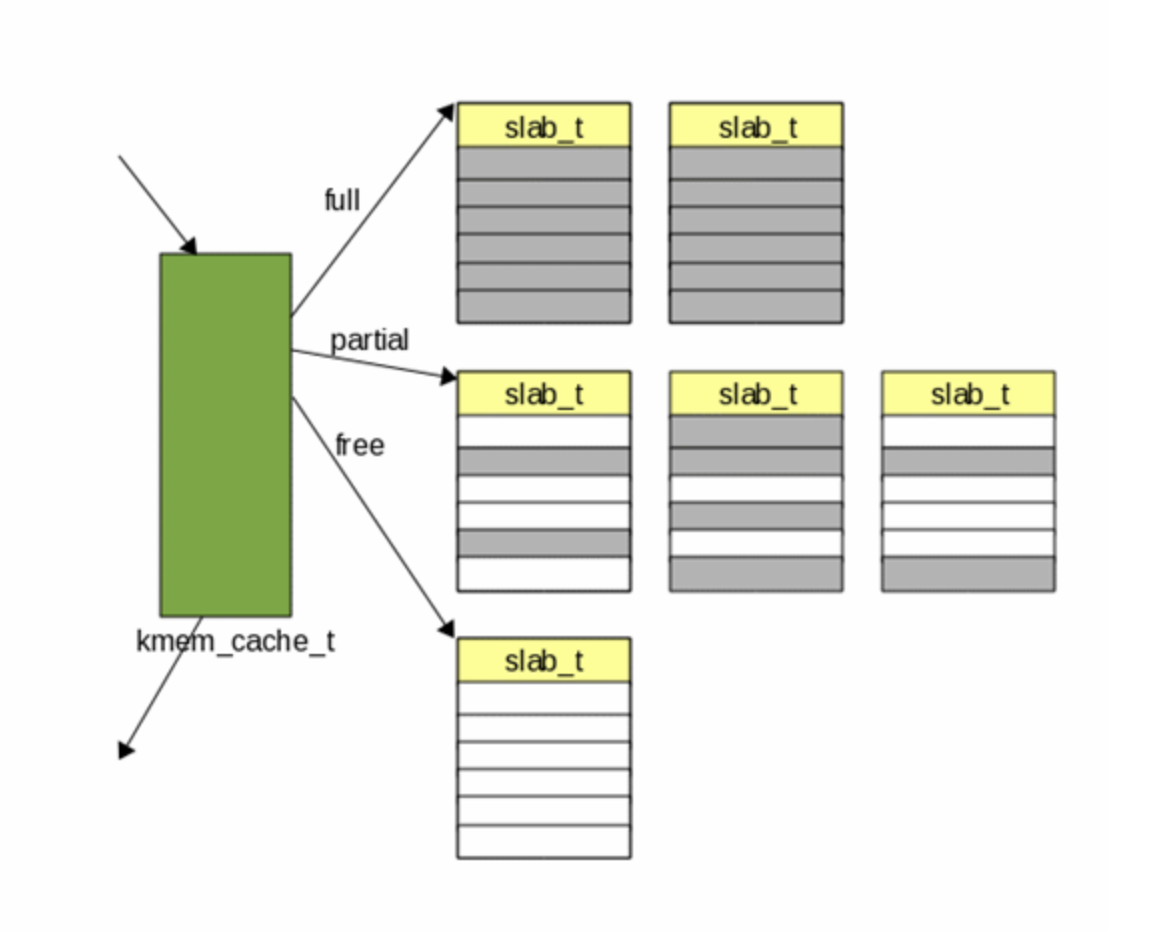
Keeping track of free objects

- kmem_bufctl array is effectively a linked list



- First free object: 3 (starts at 0)
- Next free object: 1, then 7

A cache is formed out of slabs



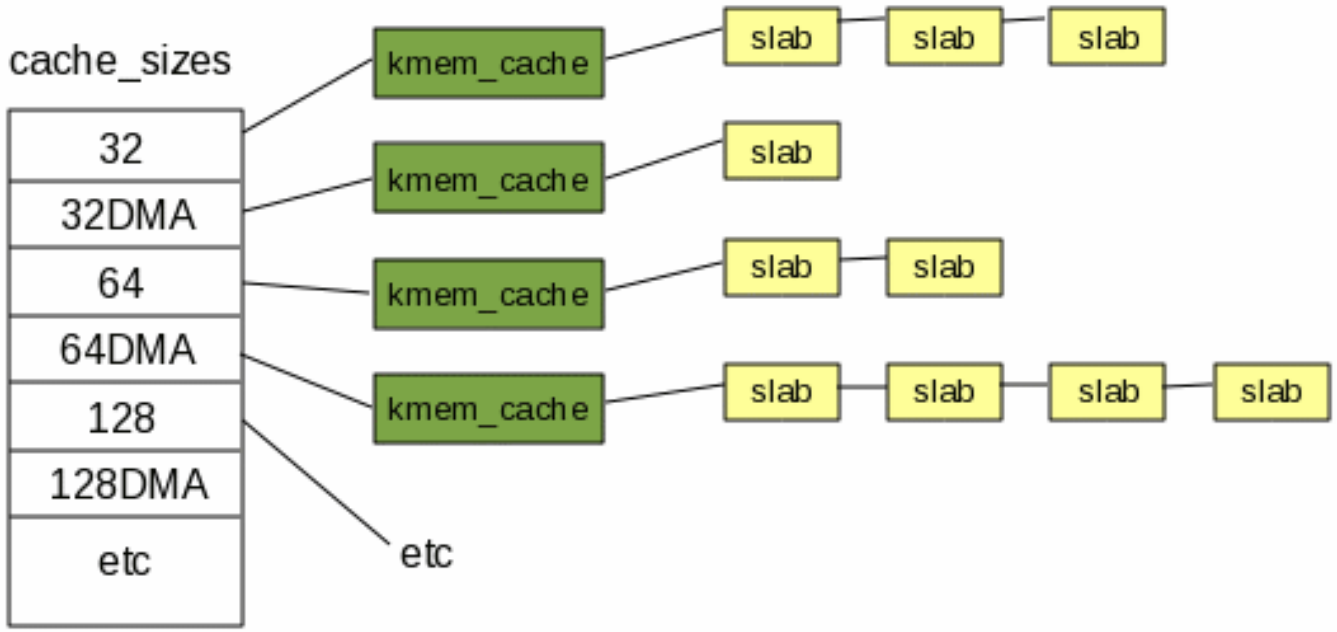
Slab is fine, but what's wrong?

Slab is fine, but what's wrong?

- We can only allocate objects of one size

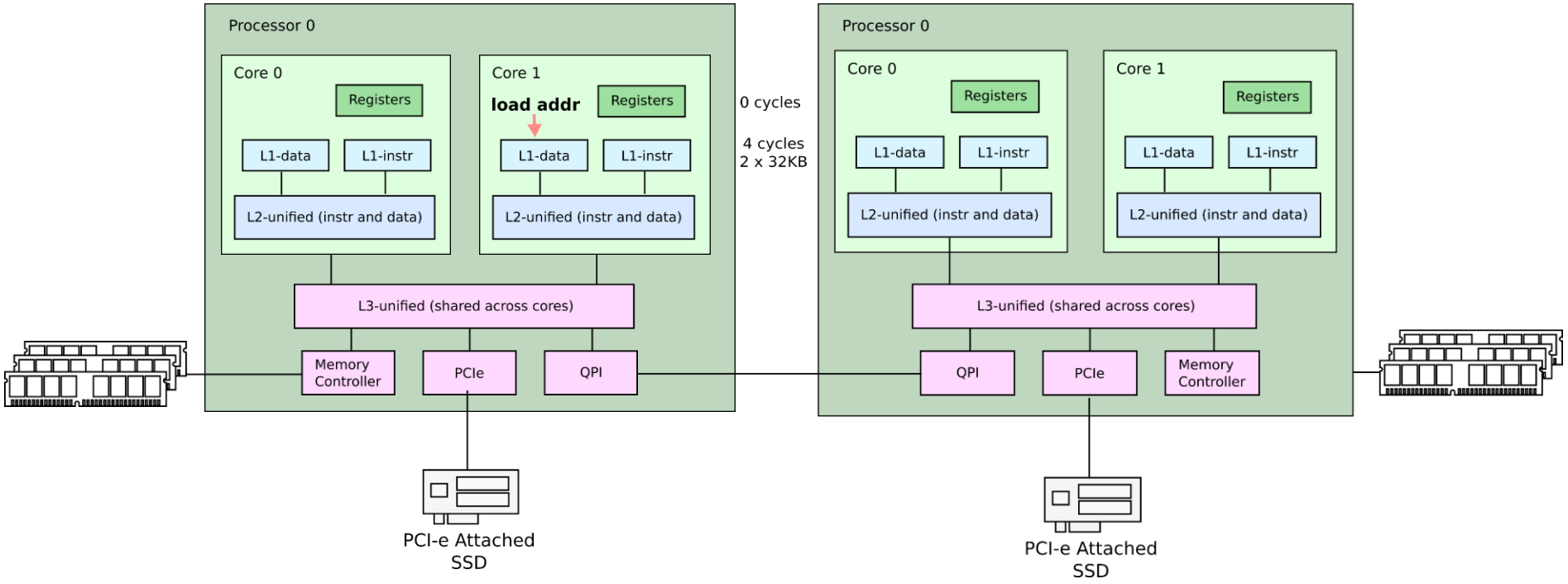
Kmalloc(): variable size objects

- A table of caches
- Size: 32, 64, 128, etc.

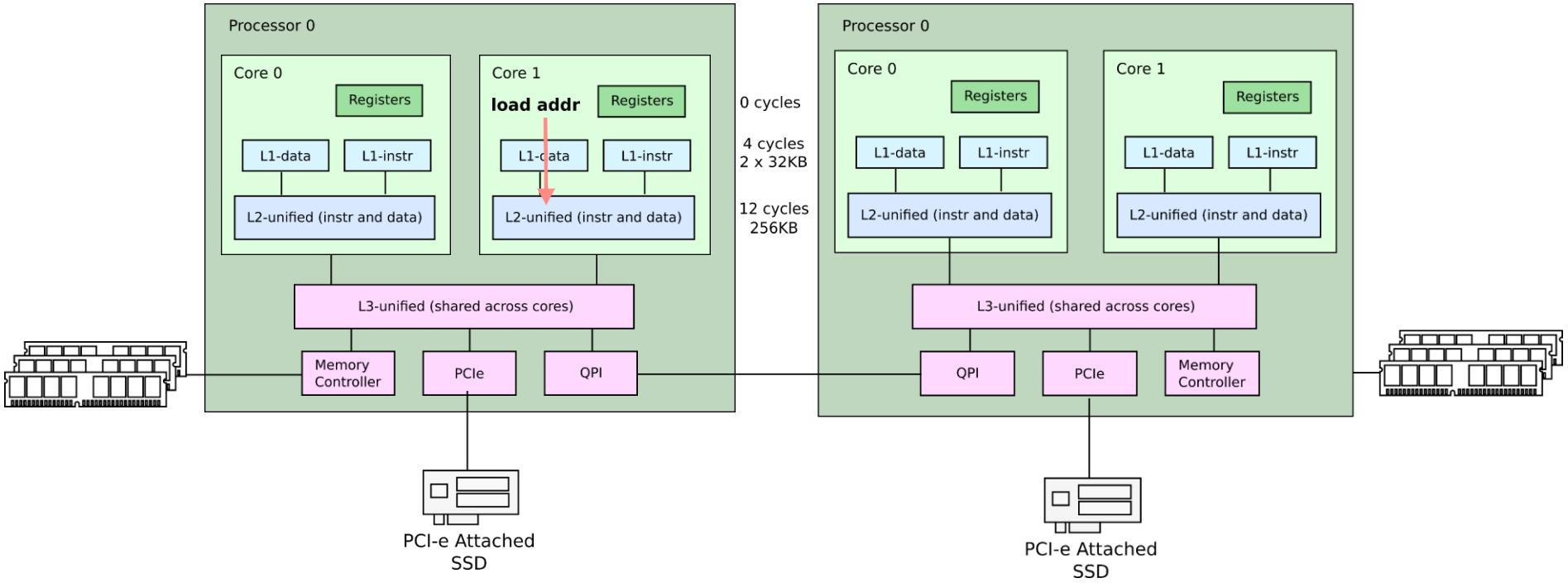


Non-Uniform Memory Access

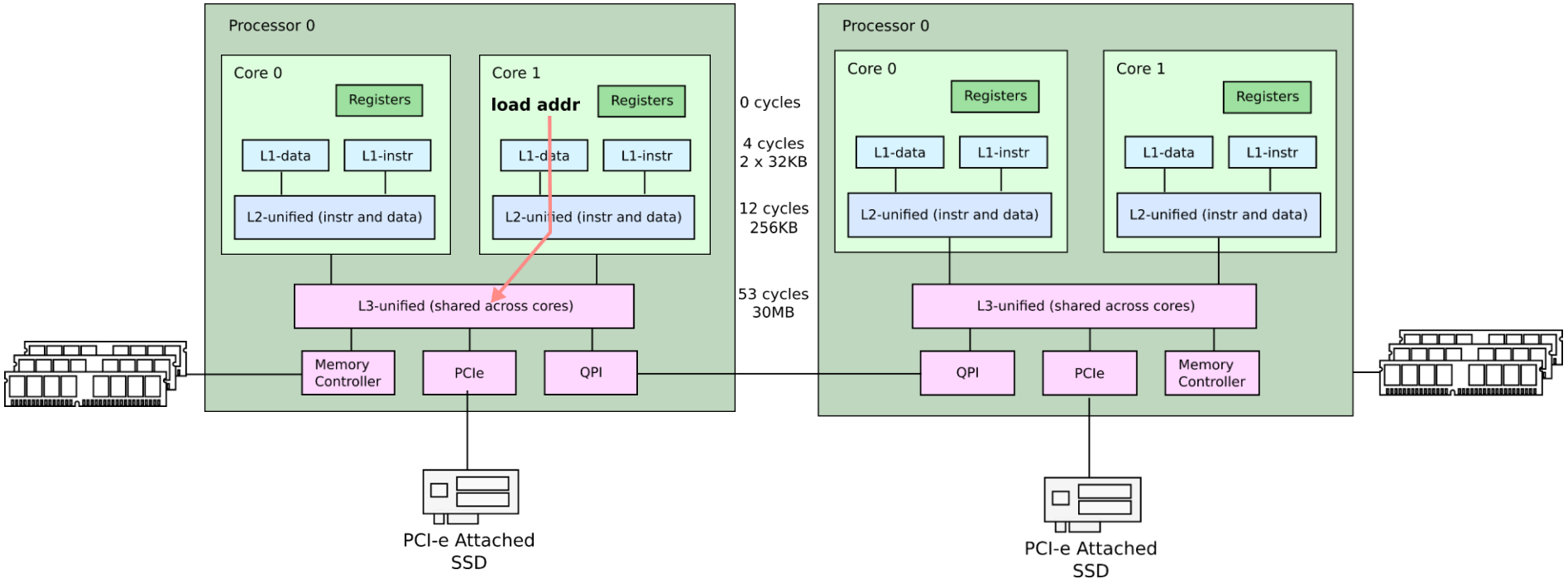
Latencies: load from local L1



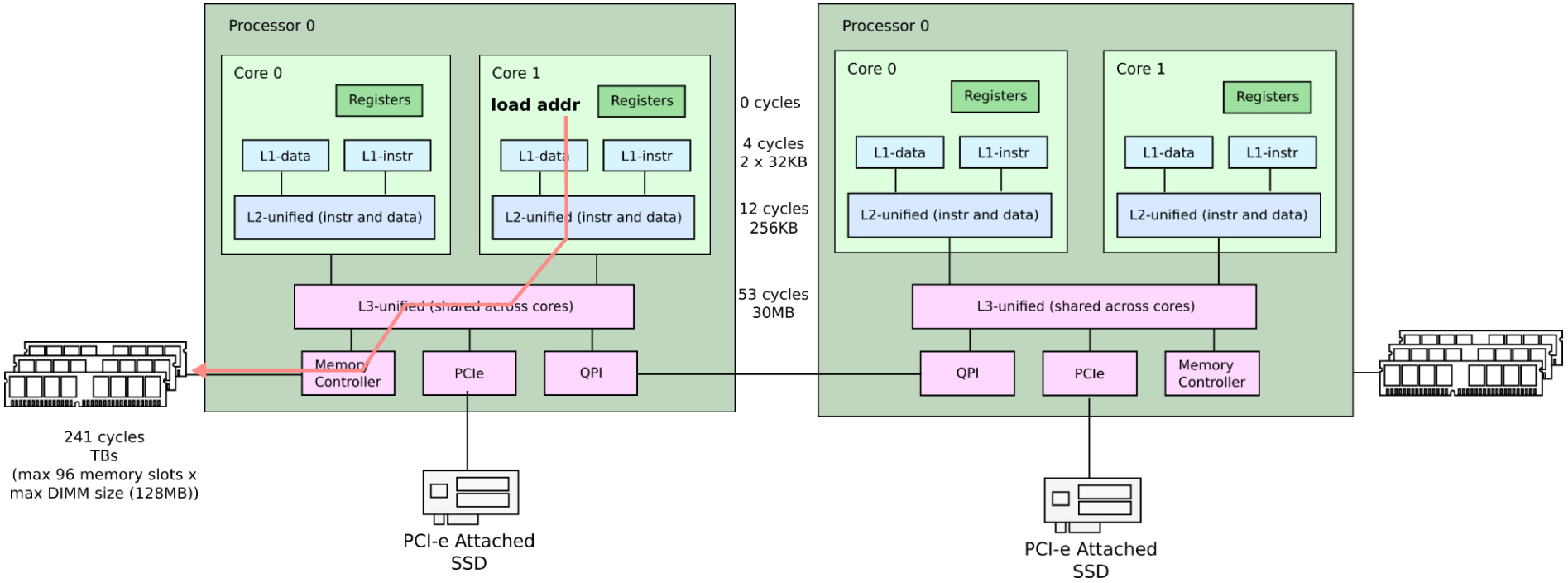
Latencies: load from local L2



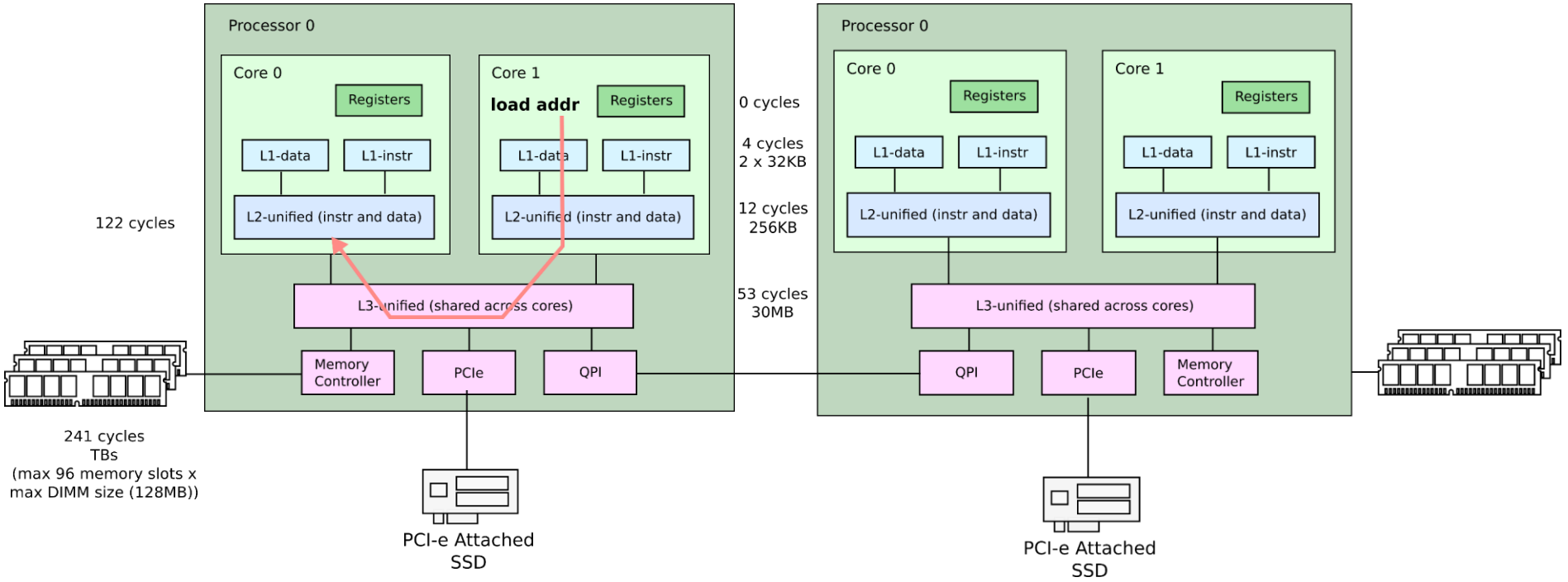
Latencies: load from local L3



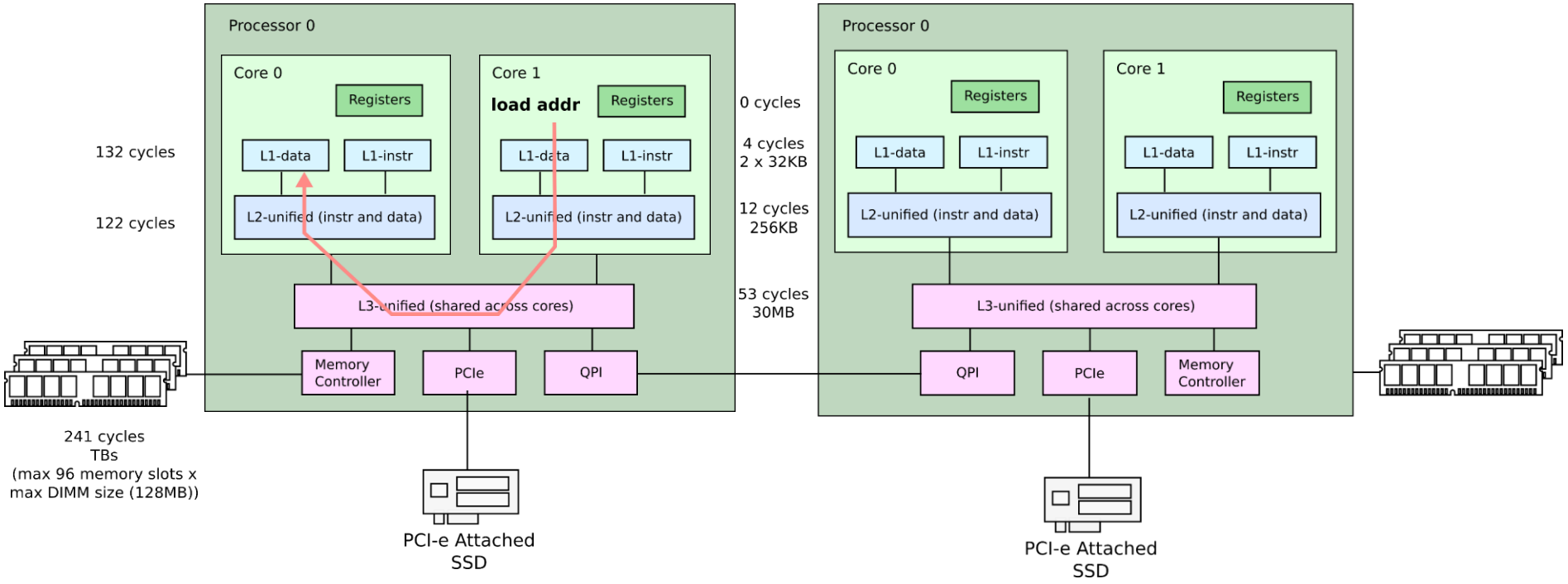
Latencies: load from local memory



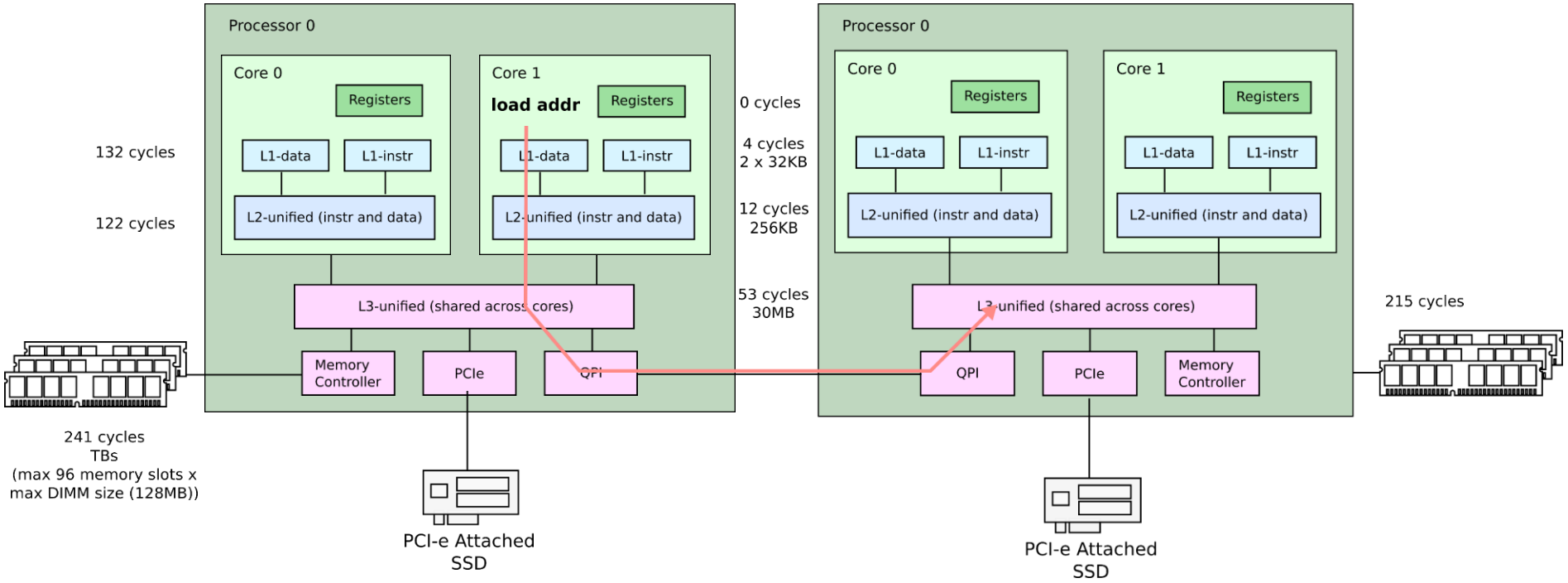
Latencies: load from same die core's L2



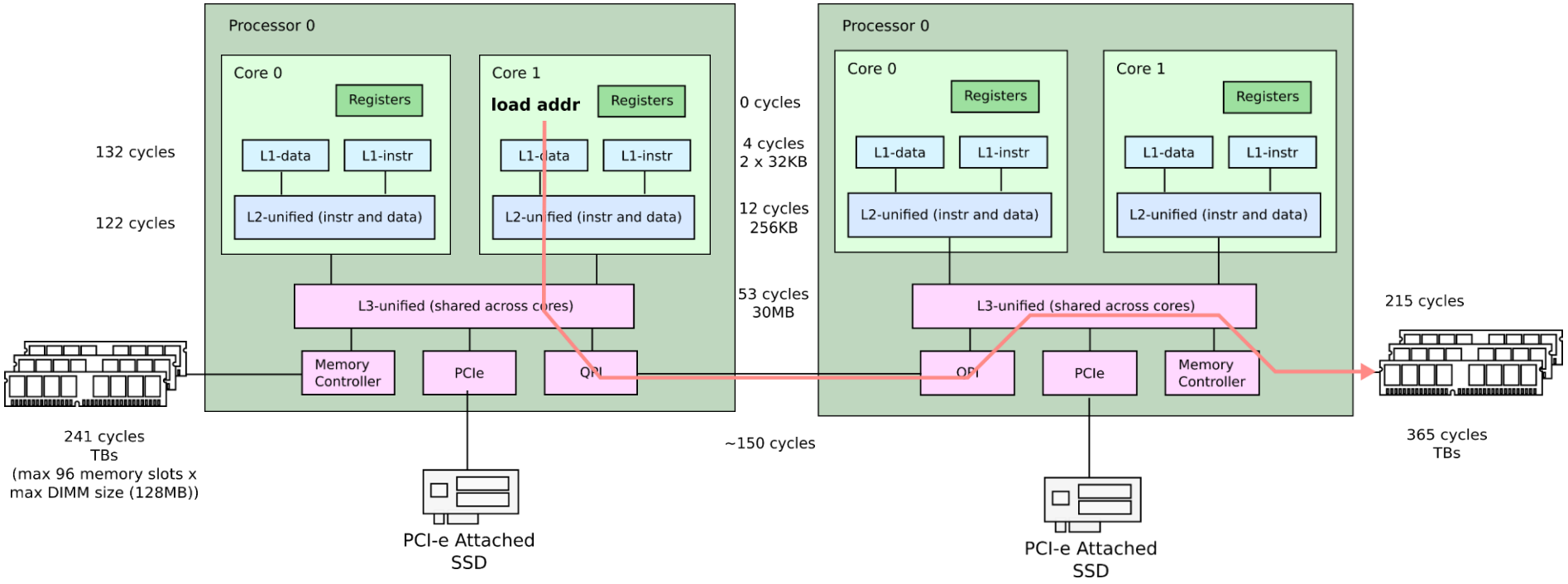
Latencies: load from same die core's L1



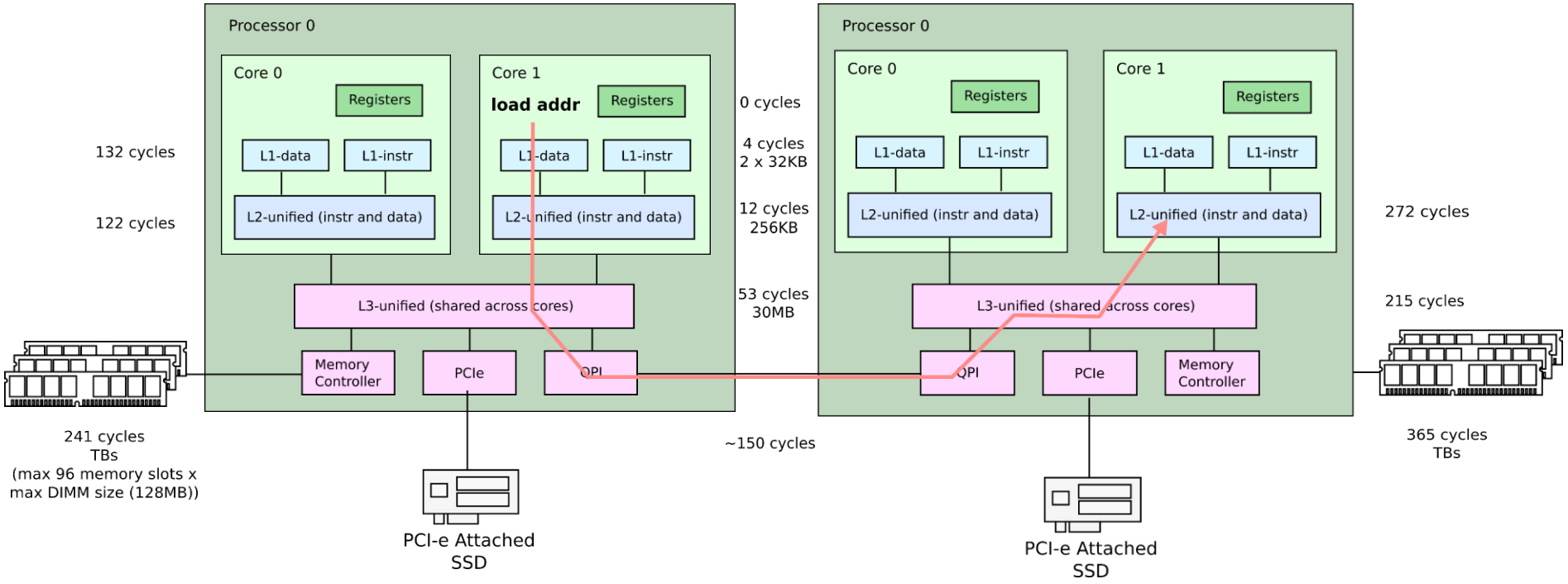
Latencies: load from remote L3



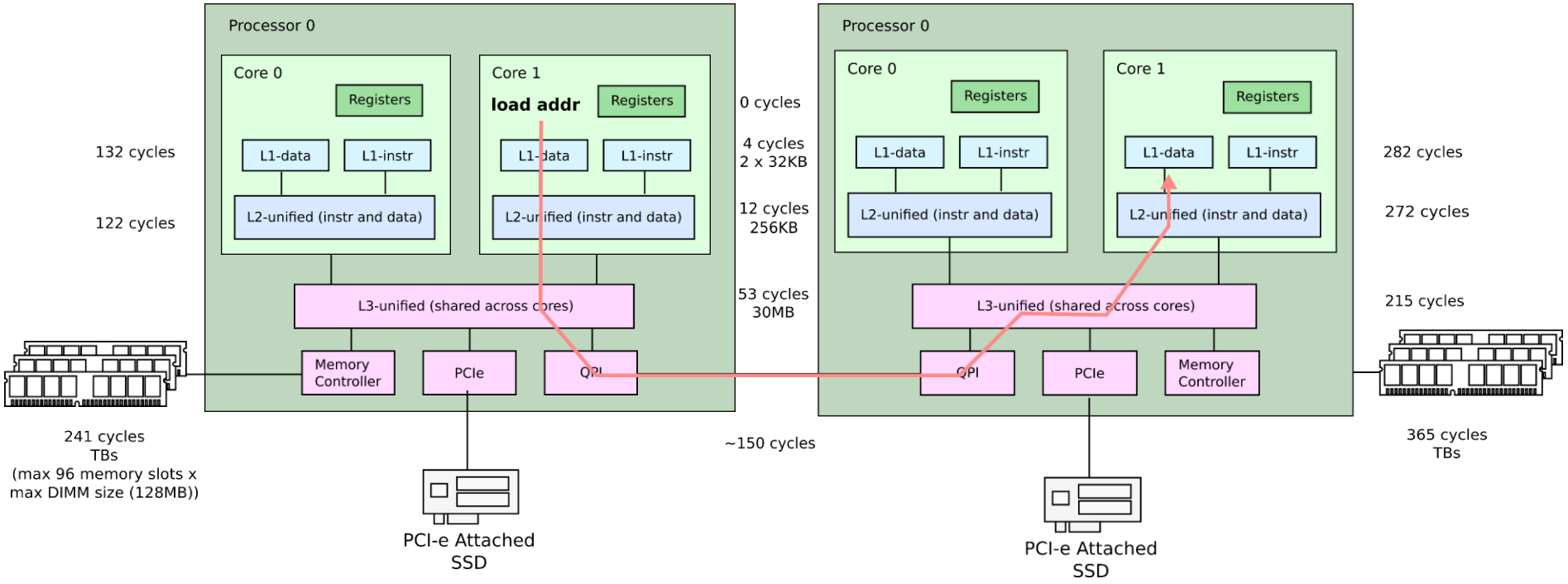
Latencies: load from remote memory



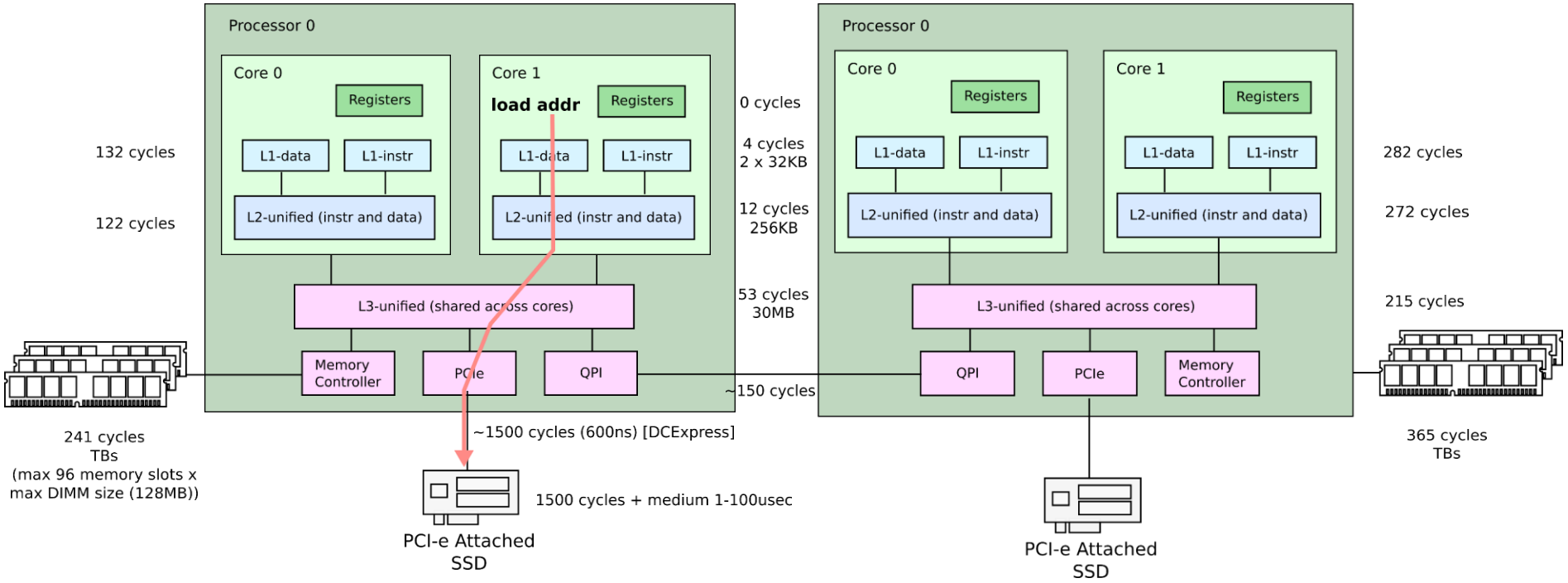
Latencies: load from remote L2



Latencies: load from remote L2



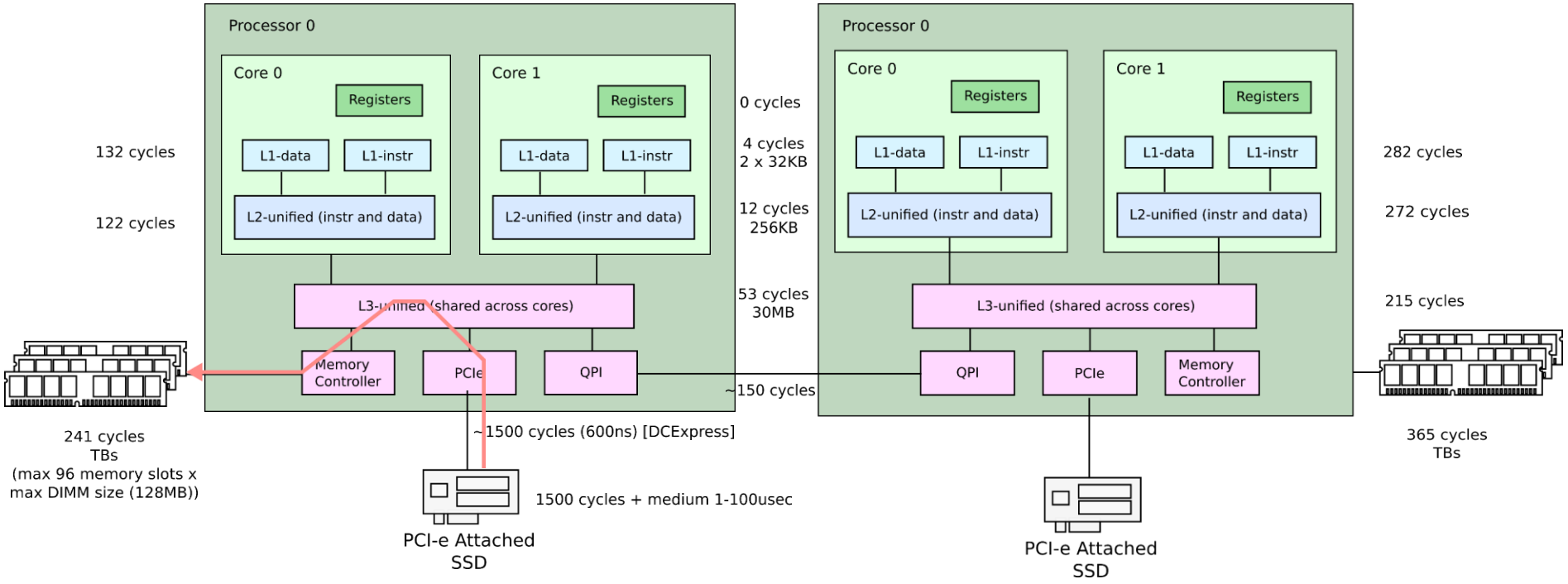
Latencies: PCIe round-trip



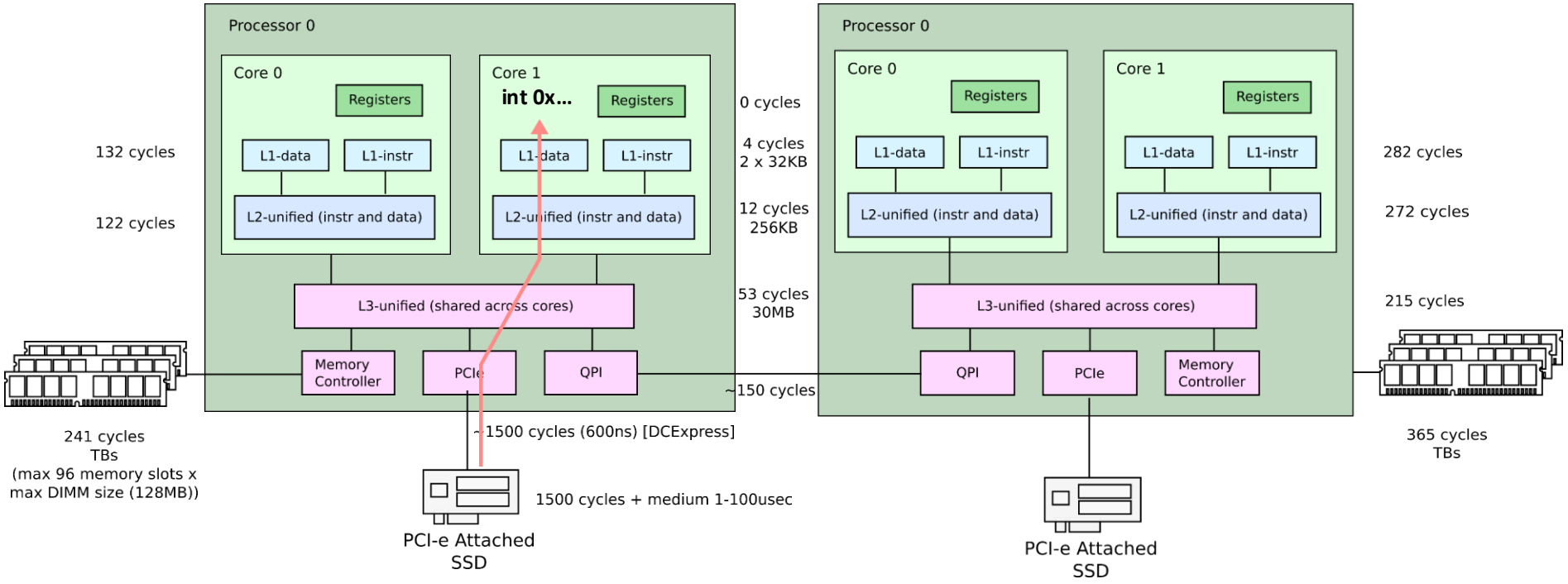
Device I/O

- Essentially just sending data to and from external devices
- Modern devices communicate over PCIe
  - Well there are other popular buses, e.g., USB, SATA (disks), etc.
  - Conceptually they are similar
- Devices can
  - Read memory
  - Send interrupts to the CPU

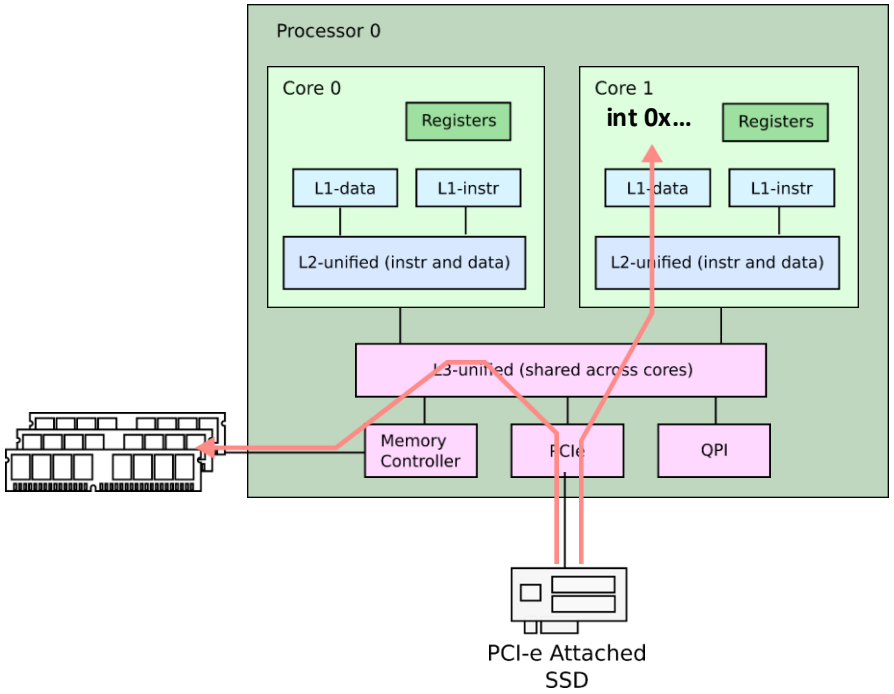
Direct memory access



Interrupts

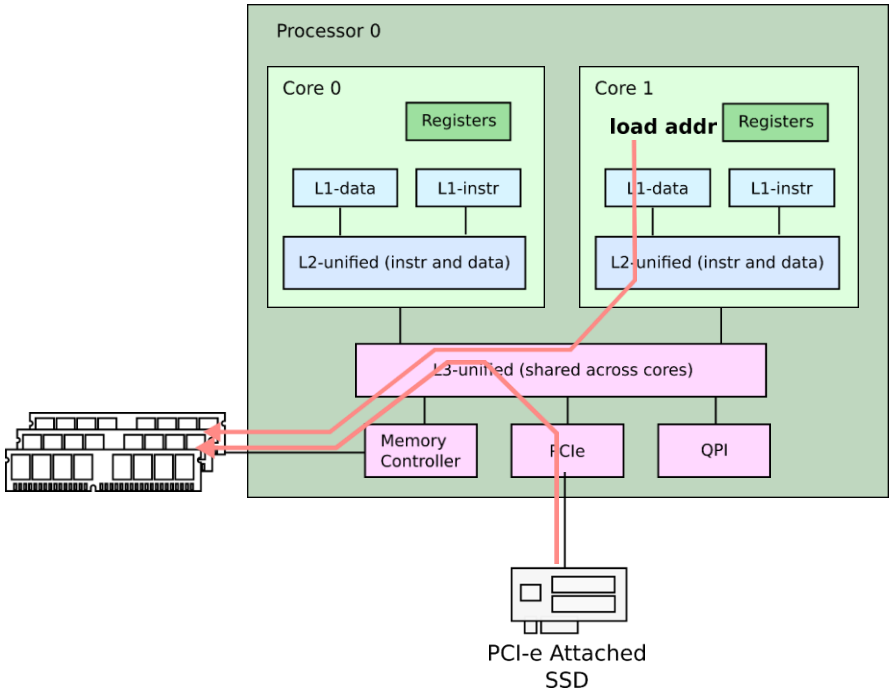

Processor 0
Core 0
Registers
L1-data
L1-instr
L2-unified (instr and data)
Core 1
int 0x...
Registers
L1-data
L1-instr
L2-unified (instr and data)
L3-unified (shared across cores)
Memory Controller
PCIe
QPI
Processor 0
Core 0
Registers
L1-data
L1-instr
L2-unified (instr and data)
Core 1
Registers
L1-data
L1-instr
L2-unified (instr and data)
L3-unified (shared across cores)
QPI
PCIe
Memory Controller
132 cycles
122 cycles
0 cycles
4 cycles
2 x 32KB
12 cycles
256KB
53 cycles
30MB
~150 cycles
282 cycles
272 cycles
215 cycles
241 cycles
TBs
(max 96 memory slots x
max DIMM size (128MB))
~1500 cycles (600ns) [DCExpress]
1500 cycles + medium 1-100usec
PCI-e Attached
SSD
PCI-e Attached
SSD
365 cycles
TBs

Device I/O



- Write incoming data in memory, e.g.,
  - Network packets
  - Disk requests, etc.
- Then raise an interrupt to notify the CPU
  - CPU starts executing interrupt handler
  - Then reads incoming packets form memory

Device I/O (polling mode)



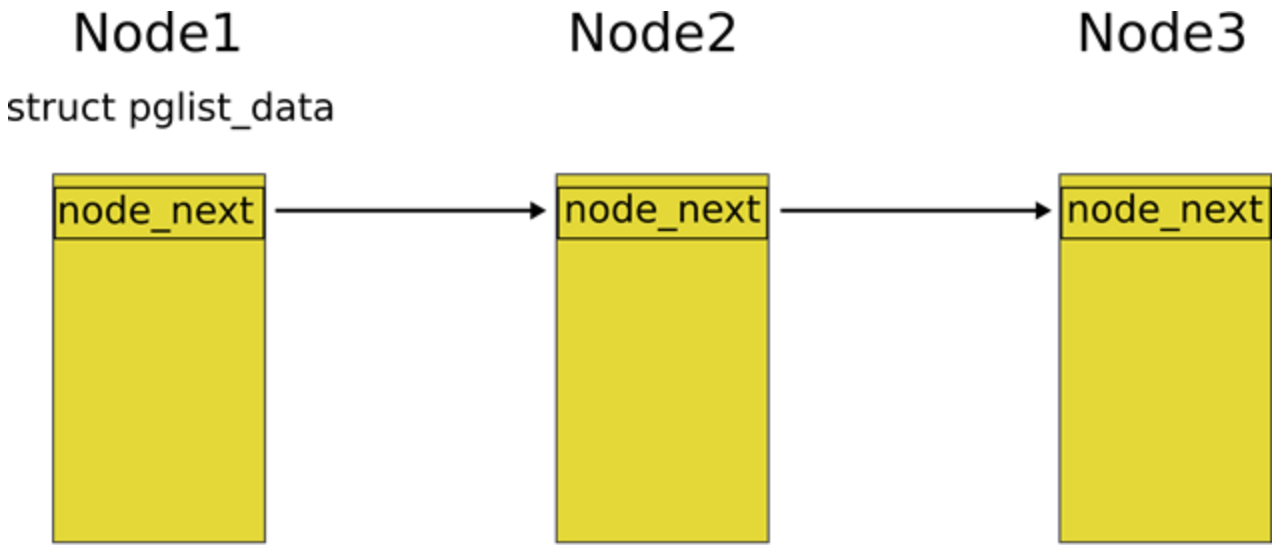
- Alternatively the CPU has to check for incoming data in memory periodically
  - Or poll
- Rationale
  - Interrupts are expensive

Uniform and non-uniform memory access

- Parts of memory can be faster than others

Nodes

- Attempt to allocate memory from the current node
- Fall back to the next node in list
  - If ran out of local memory



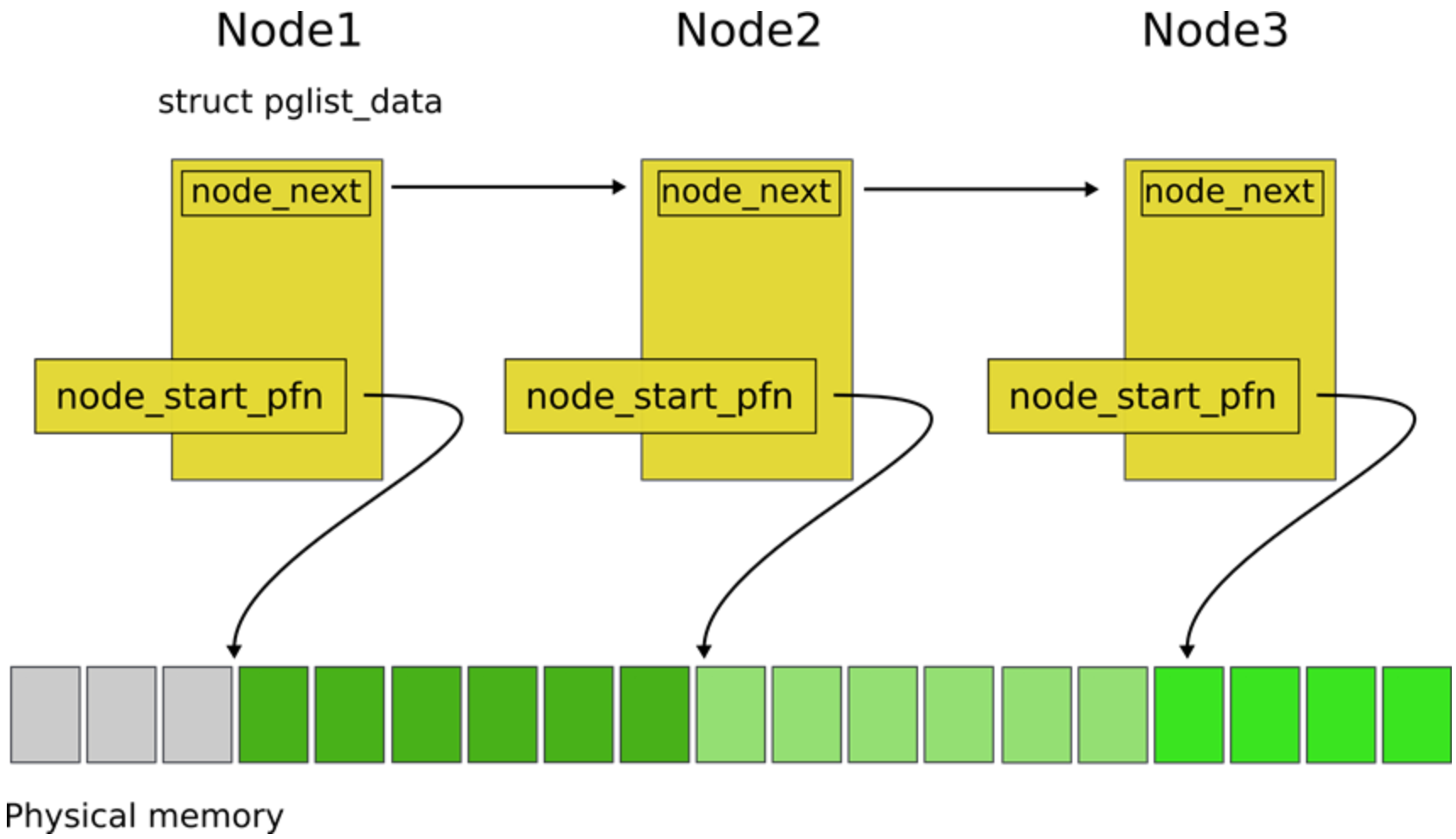
pglist_data represents a node

<mmzone.h>

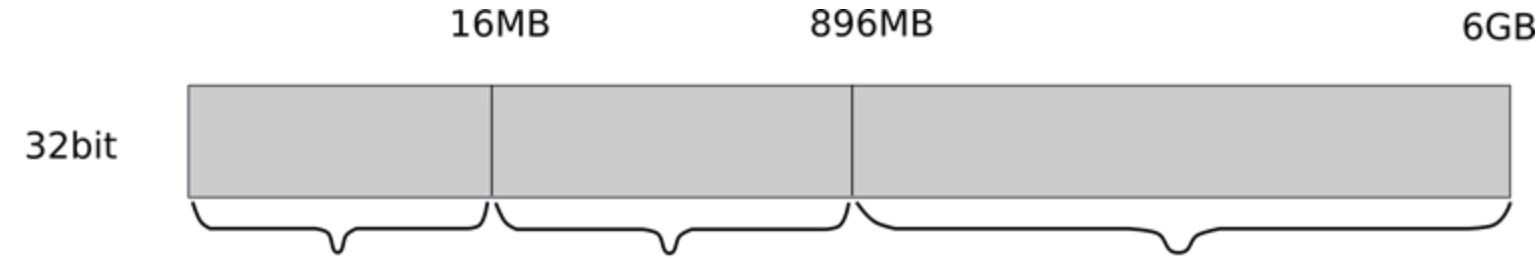
```
typedef struct pglist_data {
        struct zone node_zones[MAX_NR_ZONES];
        struct zonelist node_zonelists[MAX_ZONELISTS];
        int nr_zones;
        struct page *node_mem_map;
        struct bootmem_data *bdata;

        unsigned long node_start_pfn;
        unsigned long node_present_pages; /* total number of physical pages */
        unsigned long node_spanned_pages; /* total size of physical page
                                             range, including holes */
        int node_id;
        struct pglist_data *pgdat_next;
        wait_queue_head_t kswapd_wait;
        struct task_struct *kswapd;
        int kswapd_max_order;
} pg_data_t;
```

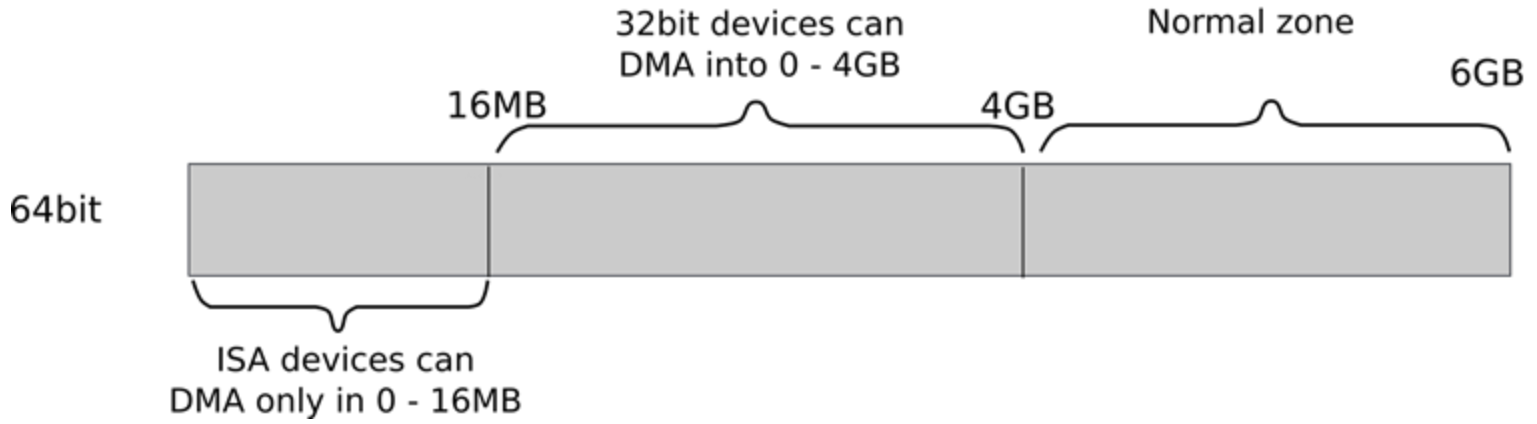
https://elixir.bootlin.com/linux/v6.10.6/source/include/linux/mmzone.h#L1277

Nodes

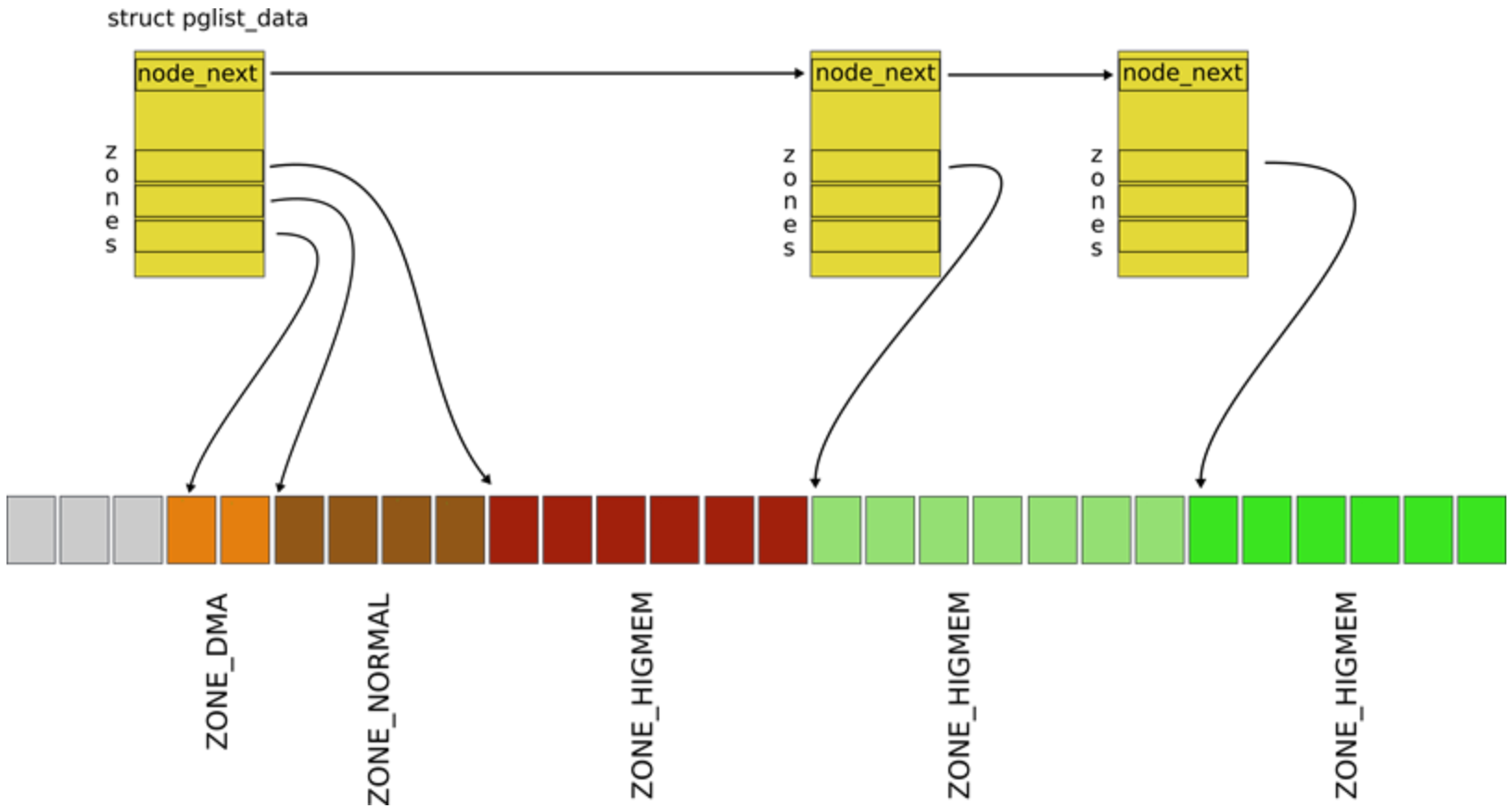

Node1
Node2
Node3
struct pglist_data
node_next
node_next
node_next
node_start_pfn
node_start_pfn
node_start_pfn
Physical memory

Zones


16MB
896MB
6GB
32bit
ISA devices can
DMA only in 0 - 16MB
Normal zone
High memory
(above 896MB on 32 bit
machines)


32bit devices can
DMA into 0 - 4GB
Normal zone
6GB
16MB
4GB
64bit
ISA devices can
DMA only in 0 - 16MB

Zones


struct pglist_data
node_next
node_next
node_next
zones
zones
zones
ZONE_DMA
ZONE_NORMAL
ZONE_HIGMEM
ZONE_HIGMEM
ZONE_HIGMEM

Thank you!