Mentoring **O**perating **S**ystem (**MentOS**)

Memory management

Created by
Enrico Fraccaroli
enrico.fraccaroli@gmail.com

Table of Contents

Physical Memory Management

Physical Memory Management

In a 32 bit-system, the 4GB address space of a RAM is divided into *page frames*. x86 processors in 32-bit mode support page sizes of 4KB, 2MB, and 4MB[1]. 4 KByte is the typical size of a page frame.

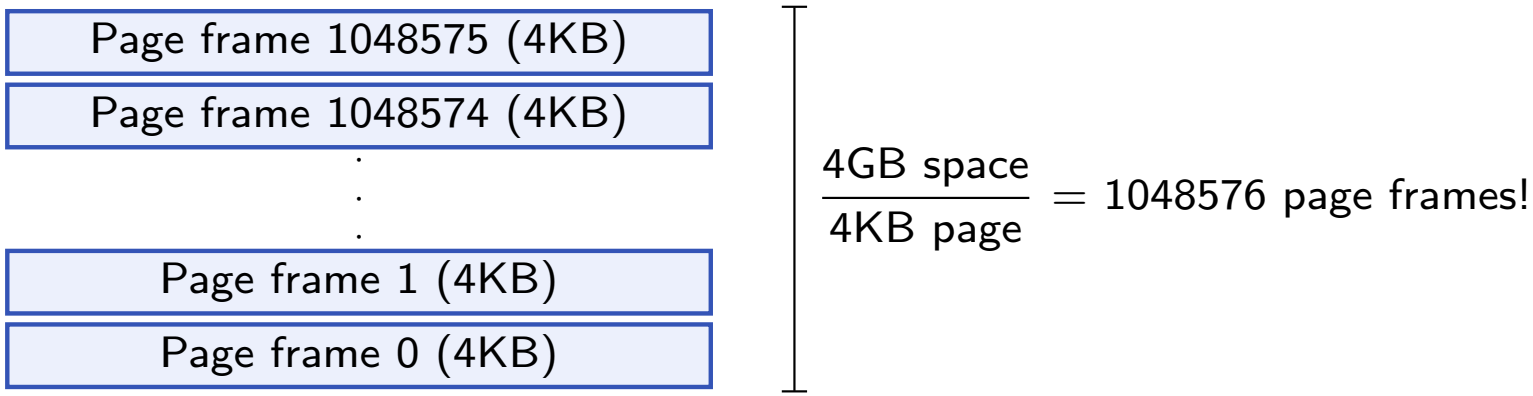


Figure: Page frames in a 4GB RAM.

To kernel, physical page frames are the basic unit of memory management.

[1] Linux and Windows map the user portion of the virtual address space in 4KB pages

Physical Memory Management

Page descriptor

Page descriptor

The kernel must keep track of the current status of each page frame. For instance, it must be able to determine if a page: is free, contains kernel code or kernel data structures, belongs to a User Mode process, etc.

The struct page_t keeps the state information of a page frame:

```
struct page_t {
    int _count;
    unsigned int private;
    struct list_head lru;
    //... continue
}
```

Page descriptor

Attributes of a page frame:

- `_count`. If it is set to -1, the corresponding page frame is free. Otherwise, the page frame is assigned to one or more processes or is used by kernel
- `private`: when page is free (used by *Buddy System*)
- `lru`: pointer to last recently used doubly linked list of pages (used by *Buddy System*)

Physical Memory Management

Zone descriptor

Zone descriptor

The Kernel partitions the physical memory into three *zones*:

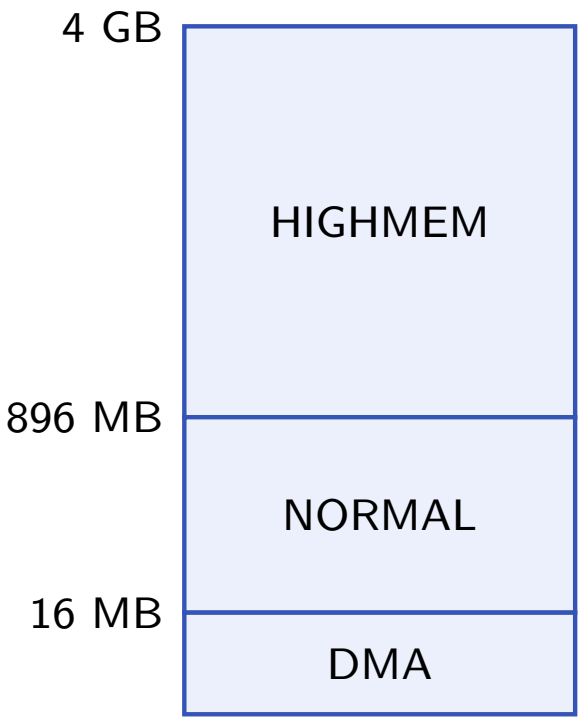


Figure: Zones in a 4GB RAM

- `ZONE_HIGHMEM` ($>$ 896 MB)
  Contains "high memory", which are page frames not permanently mapped into the kernel's address space
- `ZONE_NORMAL` (16-896 MB)
  Contains normal, regularly mapped, page frames
- `ZONE_DMA` ($<$ 16 MB)
  Contains page frames that can undergo Direct Memory Access (DMA)

Zone descriptor

Each memory zone has its own descriptor of type zone.

```
struct zone {
    unsigned long free_pages;          // Number of free pages in the zone.
    free_area_t free_area[MAX_ORDER]; // buddy blocks (see next)
    page_t * zone_mem_map;             // pointer to first page descriptor
    uint32_t zone_start_pfn;           // Index of the first page frame
    unsigned long size;                // Total size of zone in pages
    char * name;                       // Name of the zone
}
```

Zone descriptor

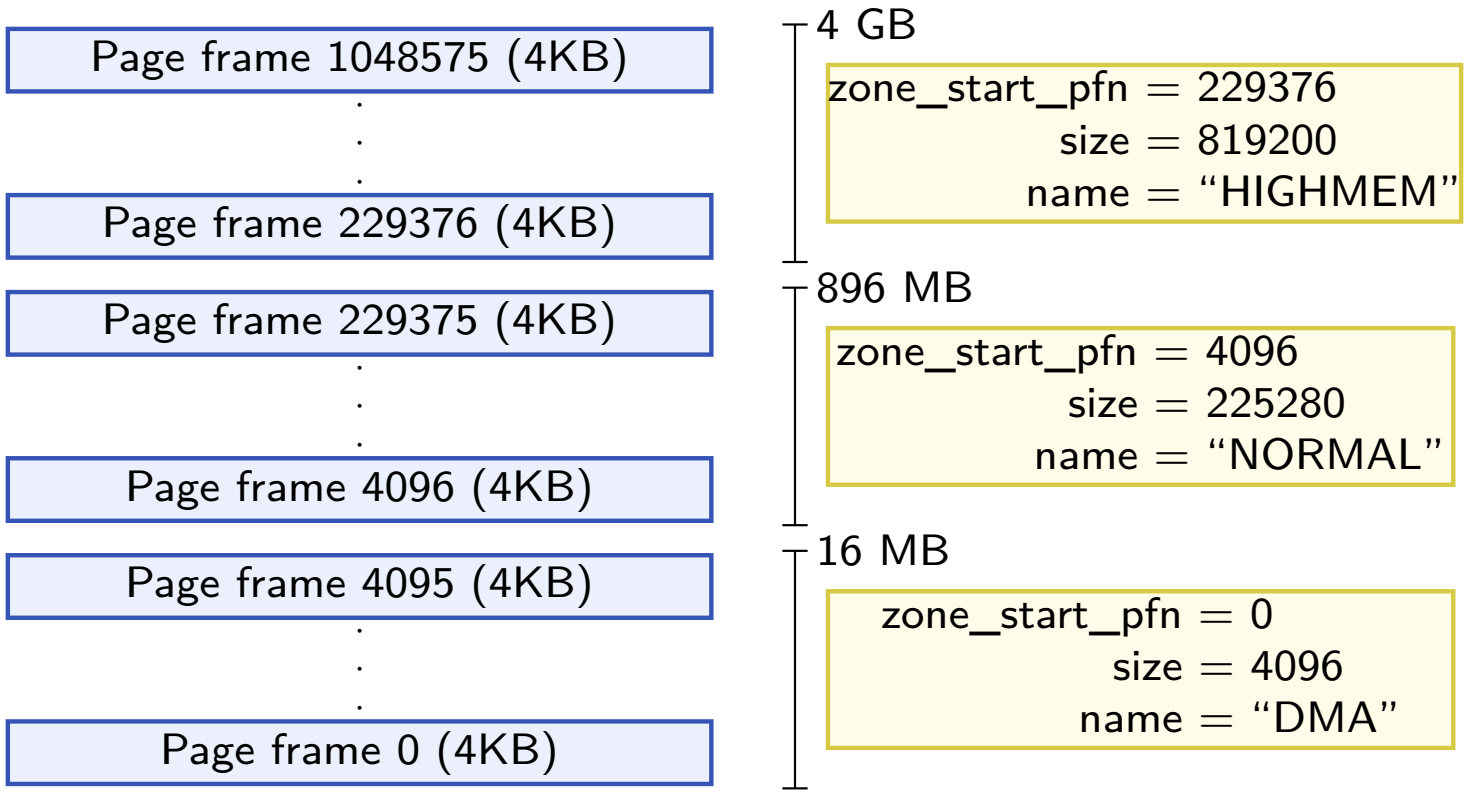


Figure: Zones in a 4GB RAM

Physical Memory Management

Zoned page frame allocator

Zoned page frame allocator

Zoned page frame allocator is the kernel subsystem handling the memory allocation/deallocation requests for groups of *contiguous* page frames.

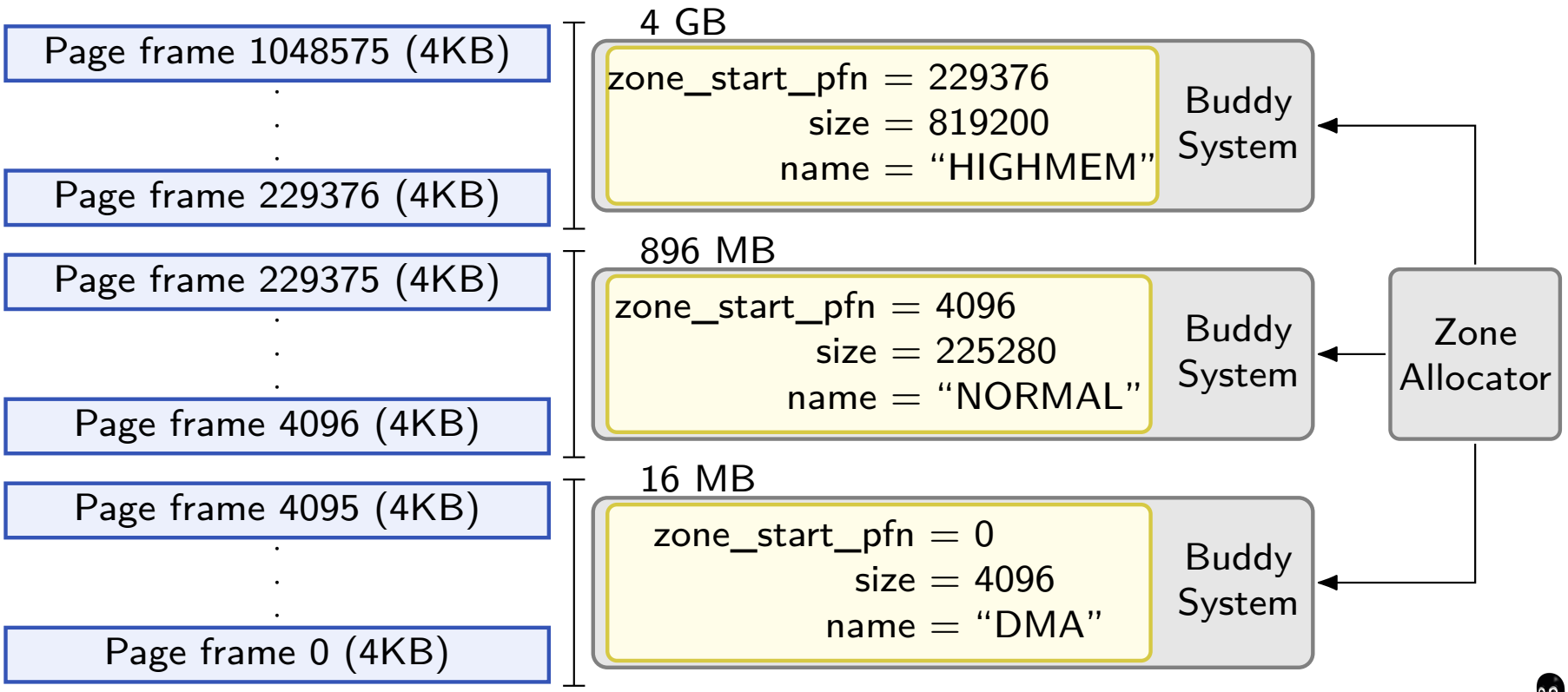


Figure: Zoned page frame allocator

Zoned page frame allocator

The Zone allocator provides the following functions to request and release page frames:

- `alloc_pages(zone, order)`
  Function used to request 2^{order} contiguous page frames from a given zone. It returns the first `page_t` of a block of 2^{order} contiguous pages, or returns NULL if the allocation failed.
- `free_pages(page, order)`
  Function used to release 2^{order} contiguous page frames of a given zone.

N.B.: Usually these functions does not receive a zone as argument, but instead a Get Free Page (GFP) flag (*e.g.*, `GFP_KERNEL`, `GFP_USER`, `GFP_DMA`, *etc.*)

Physical Memory Management

Buddy System

Buddy System

The buddy system is a robust and efficient strategy for allocating groups of contiguous page frames in power of two size.

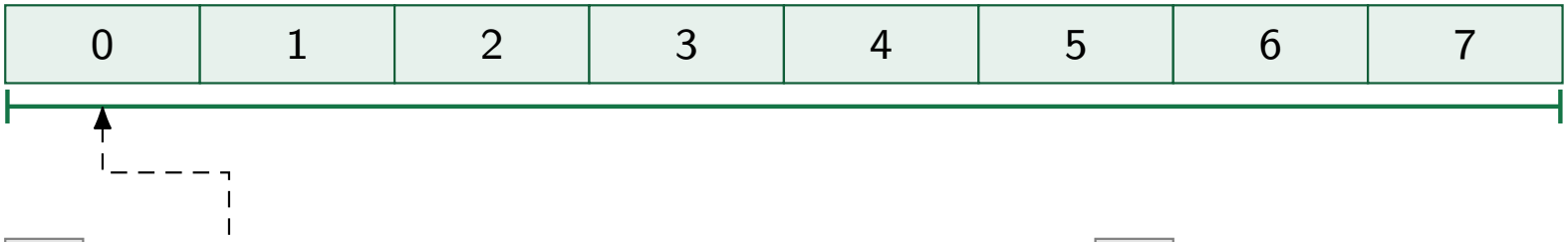
All free page frames are grouped into **11** lists of blocks that contain groups of 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, and 1024 contiguous page frames, respectively.

The largest request of 1024 page frames corresponds to a chunk of 4 MB of contiguous RAM. The smallest request is one page frame, which corresponds to a chunk of 4 KB of contiguous RAM.

Let see how the algorithm works through a simple example.

Buddy System - initial state

RAM with 8 page frames



3	{[H] → [0] } One block of 2^3 frames
2	{[H] → } Zero blocks of 2^2 frames
1	{[H] → } Zero blocks of 2^1 frames
0	{[H] → } Zero blocks of 2^0 frames

`free_area_t` list collecting
the lists of free blocks of frames

7	{free=true, order=0}
6	{free=true, order=0}
5	{free=true, order=0}
4	{free=true, order=0}
3	{free=true, order=0}
2	{free=true, order=0}
1	{free=true, order=0}
0	{free=true, order=**3**}

`page_t` array reporting
the state of each frame

Buddy System - alloc_pages (1/8)

RAM with 8 page frames

0	1	2	3	4	5	6	7

3	{[H] → [0] }
2	{[H] → }
1	{[H] → }
0	{[H] → }

A new request : A block of 2^0 frames (1 frame)

7	{free=true, order=0}
6	{free=true, order=0}
5	{free=true, order=0}
4	{free=true, order=0}
3	{free=true, order=0}
2	{free=true, order=0}
1	{free=true, order=0}
0	{free=true, order=**3**}

Buddy System - alloc_pages (2/8)

RAM with 8 page frames

0	1	2	3	4	5	6	7

3	{[H] → [0] }
2	{[H] → }
1	{[H] → }
0	{[H] → }

A new request : A block of 2^0 frames (1 frame)
Step1: search for a block big enough to satisfy the request.

7	{free=true, order=0}
6	{free=true, order=0}
5	{free=true, order=0}
4	{free=true, order=0}
3	{free=true, order=0}
2	{free=true, order=0}
1	{free=true, order=0}
0	{free=true, order=**3**}

Buddy System - alloc_pages (3/8)

RAM with 8 page frames

0	1	2	3	4	5	6	7

3	{[H] → [0] }	free block with 8 frames
2	{[H] → }	empty
1	{[H] → }	empty
0	{[H] → }	empty

A new request : A block of 2^0 frames (1 frame)
Step1: search for a block big enough to satisfy the request.
Step2: remove block from free array list.

7	{free=true, order=0}
6	{free=true, order=0}
5	{free=true, order=0}
4	{free=true, order=0}
3	{free=true, order=0}
2	{free=true, order=0}
1	{free=true, order=0}
0	{free=true, order=**3**}

Buddy System - alloc_pages (4/8)

RAM with 8 page frames

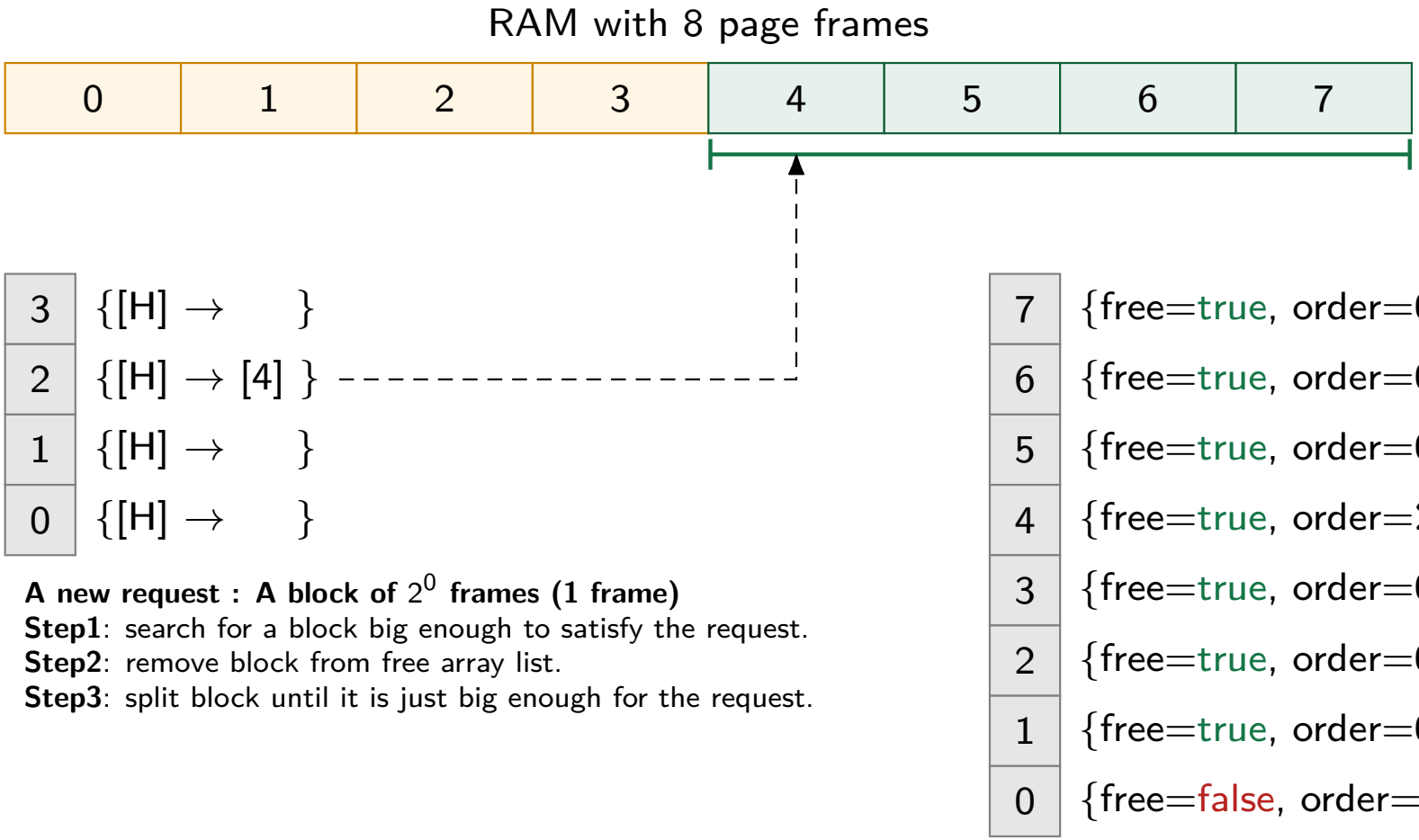
0	1	2	3	4	5	6	7

3	{[H] → }	empty - (block is taken)
2	{[H] → }	empty
1	{[H] → }	empty
0	{[H] → }	empty

A new request : A block of 2^0 frames (1 frame)
Step1: search for a block big enough to satisfy the request.
Step2: remove block from free array list.
Step3: split block until it is just big enough for the request.

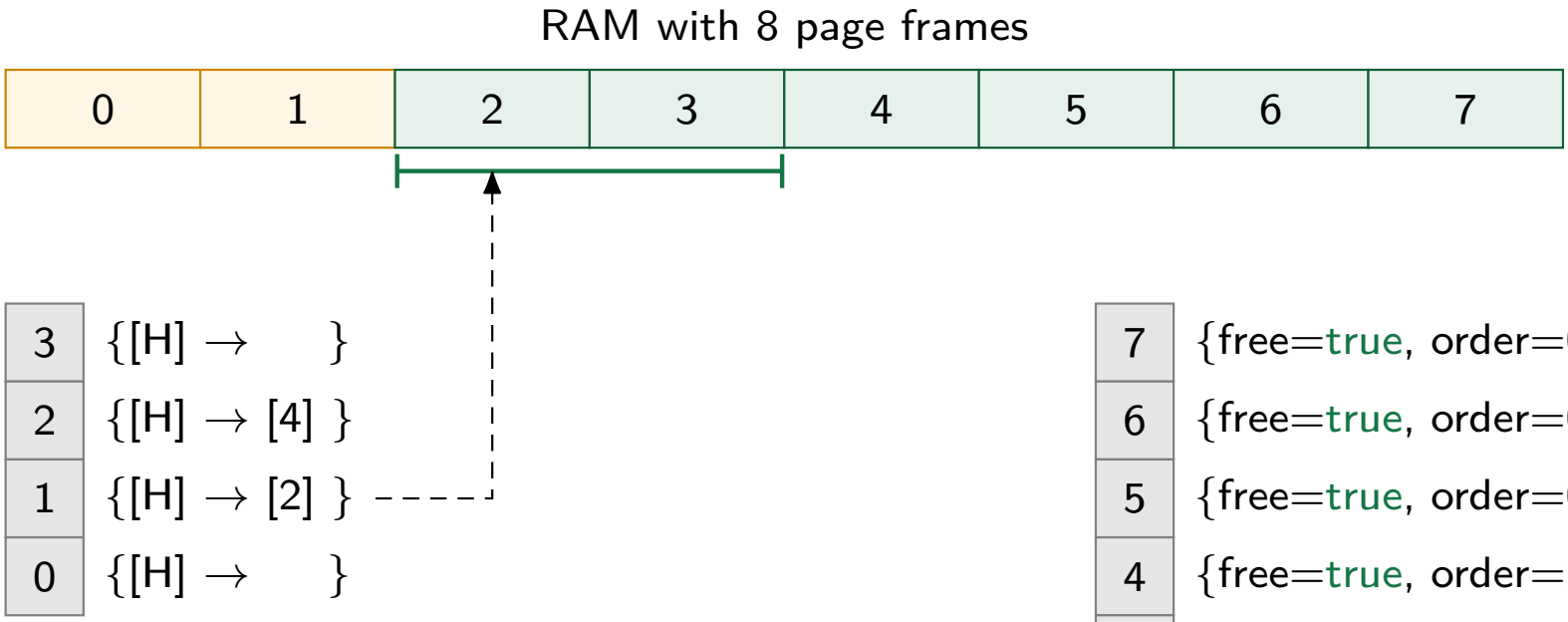
7	{free=true, order=0}
6	{free=true, order=0}
5	{free=true, order=0}
4	{free=true, order=0}
3	{free=true, order=0}
2	{free=true, order=0}
1	{free=true, order=0}
0	{free=false, order=**0**}

Buddy System - alloc_pages (5/8)



A new request : A block of 2^0 frames (1 frame)
Step1: search for a block big enough to satisfy the request.
Step2: remove block from free array list.
Step3: split block until it is just big enough for the request.

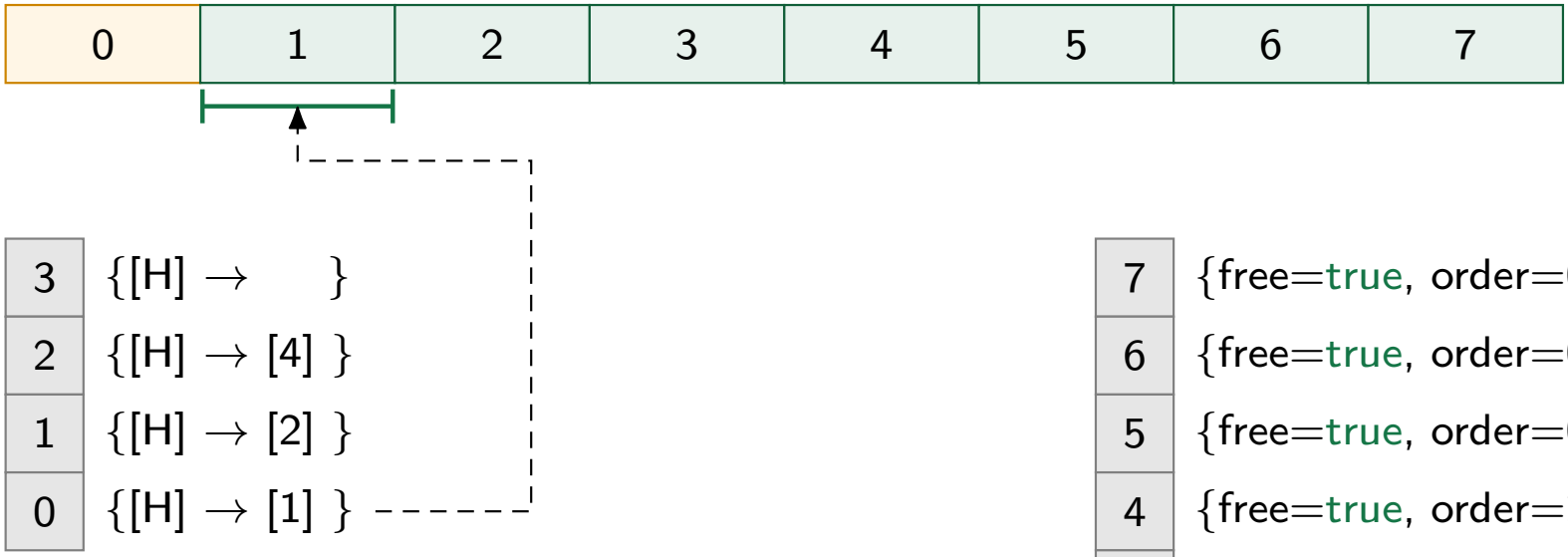
Buddy System - alloc_pages (6/8)



A new request : A block of 2^0 frames (1 frame)
Step1: search for a block big enough to satisfy the request.
Step2: remove block from free array list.
Step3: split block until it is just big enough for the request.

Buddy System - alloc_pages (7/8)

RAM with 8 page frames



A new request : A block of 2^0 frames (1 frame)
Step1: search for a block big enough to satisfy the request.
Step2: remove block from free array list.
Step3: split block until it is just big enough for the request.

7	{free=true, order=0}
6	{free=true, order=0}
5	{free=true, order=0}
4	{free=true, order=**2**}
3	{free=true, order=0}
2	{free=true, order=**1**}
1	{free=true, order=**0**}
0	{free=false, order=**0**}

Buddy System - alloc_pages (8/8)

RAM with 8 page frames

3	{[H] → }
2	{[H] → [4] }
1	{[H] → [2] }
0	{[H] → [1] }

A new request : A block of 2^0 frames (1 frame)
Step1: search for a block big enough to satisfy the request.
Step2: remove block from free array list.
Step3: split block until it is just big enough for the request.
Step4: first page of block is returned as a result.

7	{free=true, order=0}
6	{free=true, order=0}
5	{free=true, order=0}
4	{free=true, order=**2**}
3	{free=true, order=0}
2	{free=true, order=**1**}
1	{free=true, order=**0**}
0	{free=false, order=**0**}

Buddy System - Zone - alloc_pages

Require: free_area array f, Request Order ro
Ensure: Found Order of a not empty free_area, or NULL

```
1: fo = ro
2: while fo < MAX_ORDER do
3:     if !empty(f[fo]) then
4:         return fo
5:     end if
6:     fo = fo + 1
7: end while
8: return NULL
```

Algorithm 1: Search for a block big enough to satisfy the request.

Buddy System - Zone - alloc_pages

Require: free_area array f, found order of a not empty free_area fo

Ensure: A block of page frames

1: block = getFirstBlock(f[fo])
2: removeBlock(f[fo], block)
3: **return** block

Algorithm 2: Remove a block of free page frames.

Buddy System - Zone - alloc_pages

Require: free_area array f, Request Order ro, Found Order fo, Block block

1: **while** fo $>$ ro **do**
2: free_block = splitRight(block)
3: fo = fo - 1
4: addBlock(f[fo], free_block)
5: block = splitLeft(block)
6: **end while**

Algorithm 3: Split block until it is just big enough for the request.

The function *splitRight* takes in input a block, and return its right half. The function *splitLeft* takes in input a block, and return its left half.

Buddy System - free_pages

RAM with 8 page frames

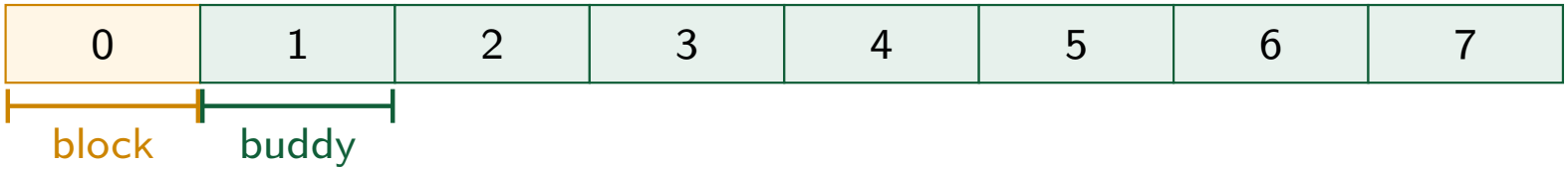
3	{[H] → }
2	{[H] → [4] }
1	{[H] → [2] }
0	{[H] → [1] }

Free page 0, which belongs to a block of order 0 (1 frame)

7	{free=true, order=0}
6	{free=true, order=0}
5	{free=true, order=0}
4	{free=true, order=2}
3	{free=true, order=0}
2	{free=true, order=1}
1	{free=true, order=0}
0	{free=false, order=0}

Buddy System - free_pages

RAM with 8 page frames



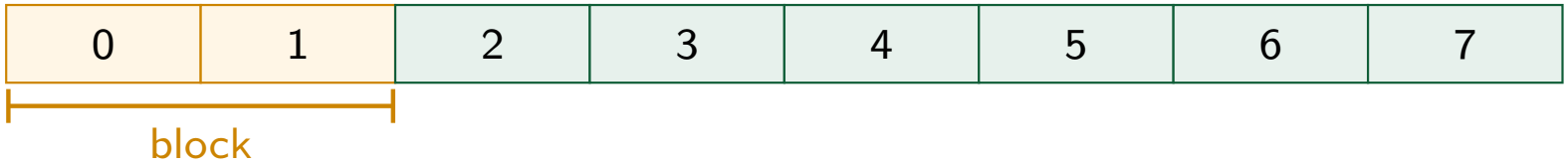
3	{[H] → }
2	{[H] → [4] }
1	{[H] → [2] }
0	{[H] → [1] }

7	{free=true, order=0}
6	{free=true, order=0}
5	{free=true, order=0}
4	{free=true, order=2}
3	{free=true, order=0}
2	{free=true, order=1}
1	{free=true, order=0}
0	{free=false, order=0}

Free page 0, which belongs to a block of order 0 (1 frame)
Step 1: check if the buddy block of the given one is free.

Buddy System - free_pages

RAM with 8 page frames

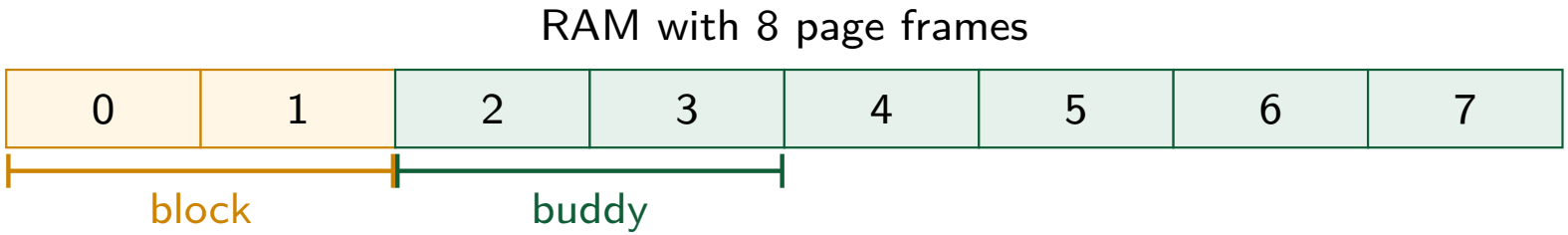


3	{[H] → }
2	{[H] → [4] }
1	{[H] → [2] }
0	{[H] → }

Free page 0, which belongs to a block of order 0 (1 frame)
Step 1: check if the buddy block of the given one is free.
Step 1.1: if found, merge block with its buddy block

7	{free=true, order=0}
6	{free=true, order=0}
5	{free=true, order=0}
4	{free=true, order=2}
3	{free=true, order=0}
2	{free=true, order=1}
1	{free=true, order=**0**}
0	{free=false, order=**1**}

Buddy System - free_pages



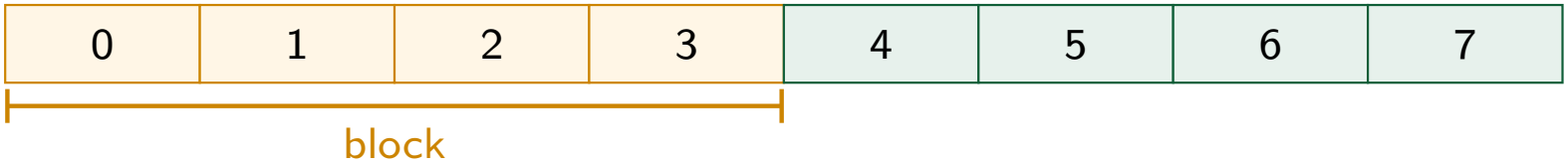
3	{[H] → }
2	{[H] → [4] }
1	{[H] → [2] }
0	{[H] → }

Free page 0, which belongs to a block of order 0 (1 frame)
Step 1: check if the buddy block of the given one is free.
Step 2: if found, merge block with its buddy block
Step 3: repeat from **Step1**

7	{free=true, order=0}
6	{free=true, order=0}
5	{free=true, order=0}
4	{free=true, order=2}
3	{free=true, order=0}
2	{free=true, order=1}
1	{free=true, order=**0**}
0	{free=false, order=**1**}

Buddy System - free_pages

RAM with 8 page frames

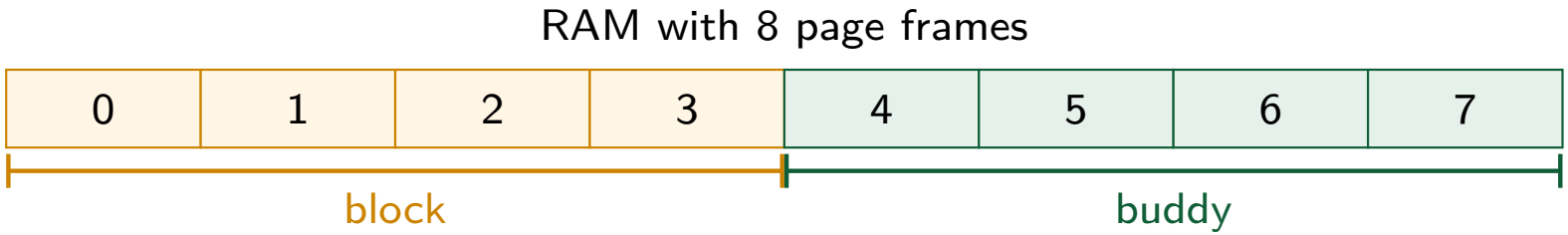


3	{[H] → }
2	{[H] → [4] }
1	{[H] → }
0	{[H] → }

Free page 0, which belongs to a block of order 0 (1 frame)
Step 1: check if the buddy block of the given one is free.
Step 2: if found, merge block with its buddy block
Step 3: repeat from **Step1**

7	{free=true, order=0}
6	{free=true, order=0}
5	{free=true, order=0}
4	{free=true, order=2}
3	{free=true, order=0}
2	{free=true, order=**0**}
1	{free=true, order=**0**}
0	{free=false, order=**2**}

Buddy System - free_pages



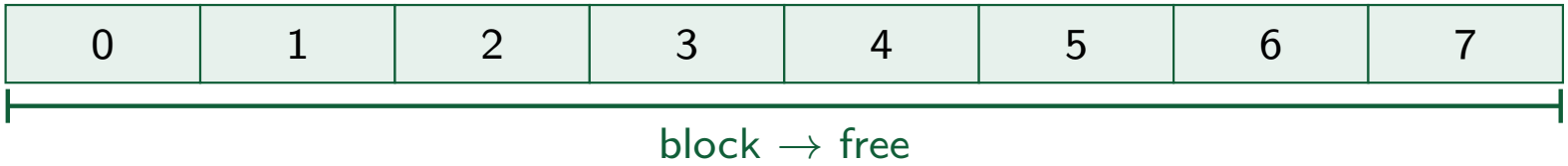
3	{[H] → }
2	{[H] → [4] }
1	{[H] → }
0	{[H] → }

Free page 0, which belongs to a block of order 0 (1 frame)
Step 1: check if the buddy block of the given one is free.
Step 2: if found, merge block with its buddy block
Step 3: repeat from **Step1**

7	{free=true, order=0}
6	{free=true, order=0}
5	{free=true, order=0}
4	{free=true, order=2}
3	{free=true, order=0}
2	{free=true, order=**0**}
1	{free=true, order=**0**}
0	{free=false, order=**2**}

Buddy System - free_pages

RAM with 8 page frames



3	{[H] → [0] }
2	{[H] → }
1	{[H] → }
0	{[H] → }

Free page 0, which belongs to a block of order 0 (1 frame)
Step 1: check if the buddy block of the given one is free.
Step 2: if found, merge block with its buddy block
Step 3: repeat from **Step1**
Step 4: add block to free area list.

7	{free=true, order=0}
6	{free=true, order=0}
5	{free=true, order=0}
4	{free=true, order=**0**}
3	{free=true, order=0}
2	{free=true, order=**0**}
1	{free=true, order=**0**}
0	{free=true, order=**3**}

Buddy System - Zone - free_pages

Require: free_area array f, Block b, Order o

```
1:  while o < MAX_ORDER - 1 do
2:      buddy = getBuddy(b, o)
3:      if !free(buddy) | order(buddy) ≠ o then
4:          break;
5:      end if
6:      removeBlock(f[o], buddy)
7:      if buddy < b then
8:          b = buddy
9:      end if
10:     o = o + 1
11: end while
12: addBlock(f[o], b)
```

Algorithm 4: Search for a block big enough to satisfy the request.

Virtual Memory Management

Virtual Memory Management

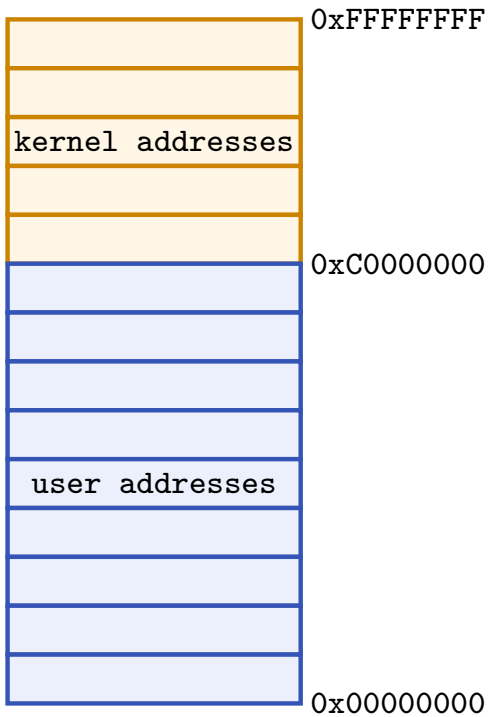


Figure: Kernel and User space in a Virtual Memory.

The Kernel applies Virtual Memory to maps virtual addresses to physical addresses. Advantages:

- RAM can be virtually split in kernel and user space;
- each single page frame can have different access permissions;
- *each process* have its memory mapping;
- a process can only access a subset of the available physical memory;
- a process can be relocatable.

How does processor translate a virtual address into a physical address?

Virtual Memory Management

The MMU and TLB

Memory Management Unit

Memory Management Unit is the hardware component mapping virtual addresses into physical address. Advantages: mapping is performed in hardware, thus no performance penalty, same CPU instructions used for access RAM and mapped hardware.

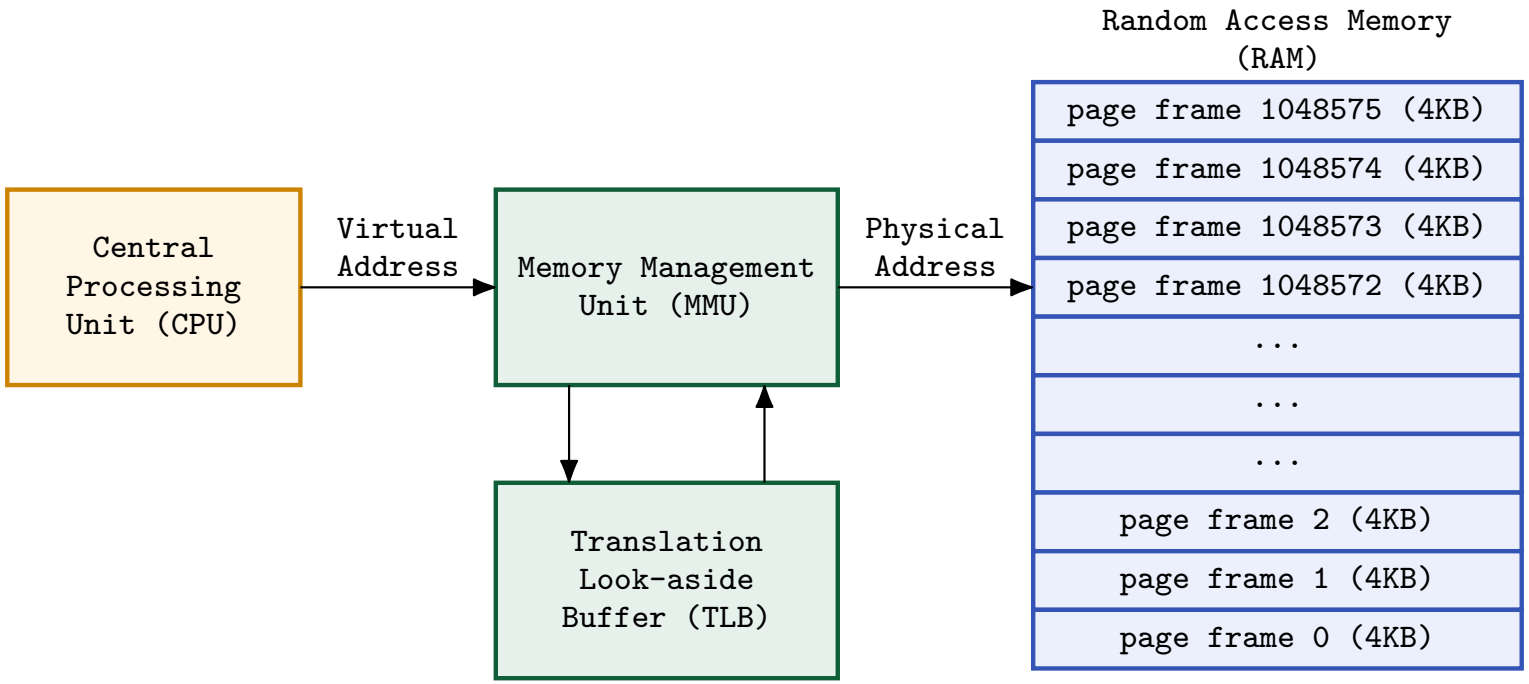


Figure: Memory Management Unit (MMU)

Memory Management Unit

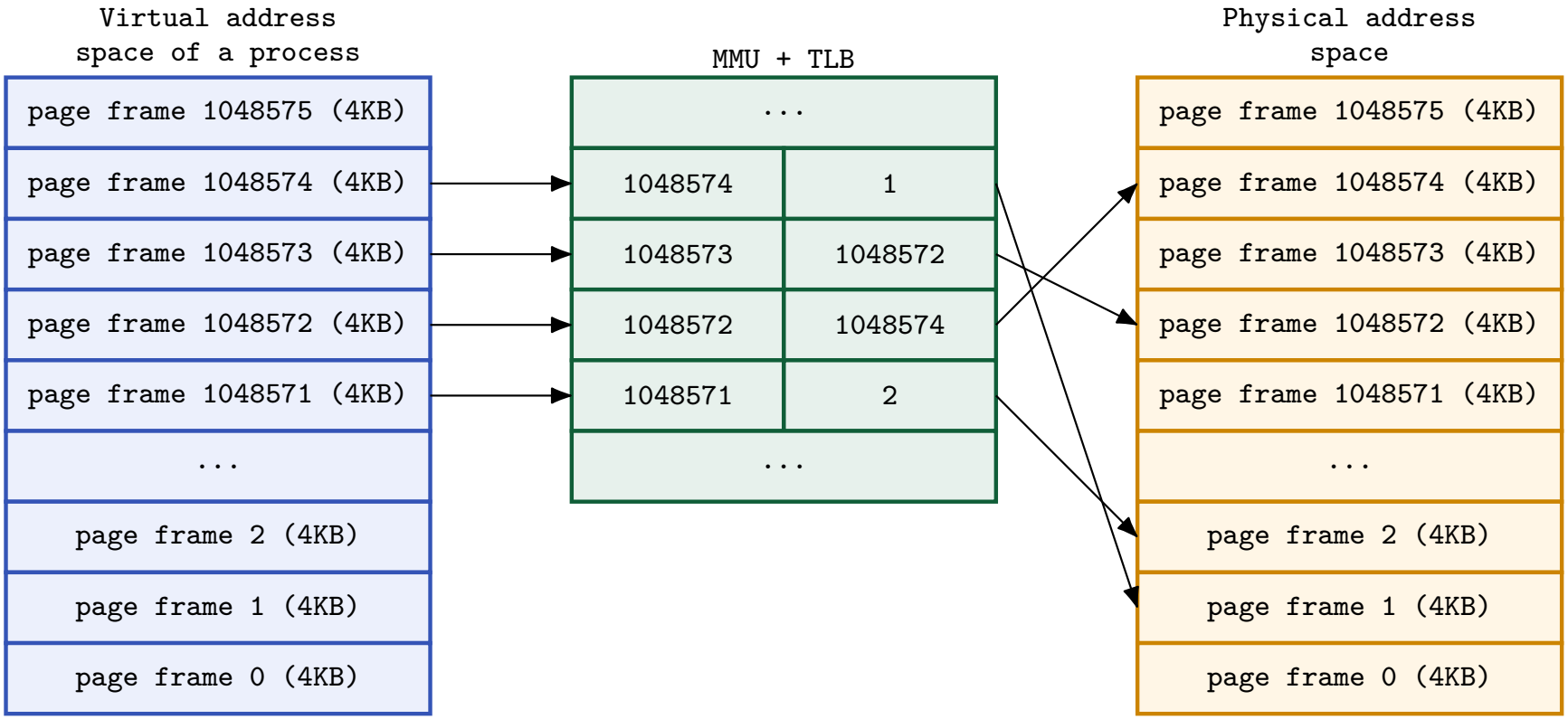


Figure: Translation Look-aside Buffer (TLB)

How does the Kernel keep track of the mapping between one process's virtual page to its corresponding page frame?

From Virtual to physical

For each process virtual page, the Kernel keeps a corresponding Page Table Entry (PTE).

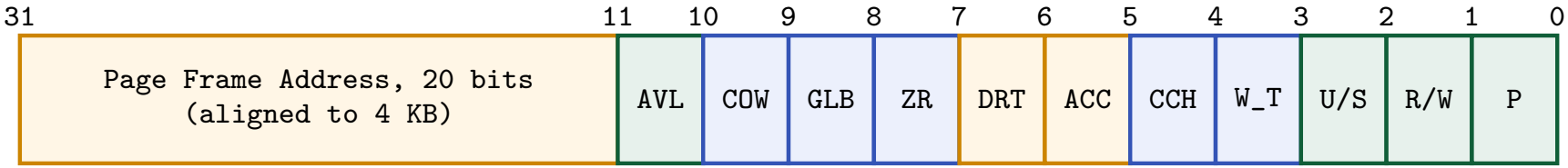


Figure: Page Table Entry.

- P when set, the page is in physical memory.
- R/W when set, the page is in read/write mode.
- U/S when set, the page can be accessed by all.
- ACC when set, the page was accessed.
- DRT when set, the page is dirty.

How does the Kernel keep track of the mapping of all process's virtual pages to their corresponding page frames?

From Virtual to physical

For each process, the Kernel organize its PTEs in a two level hierarchical data structure:
First level: a Page Directory collecting 1024 addresses to Page Table.
Second level: a Page Table collects 1024 page table entries.

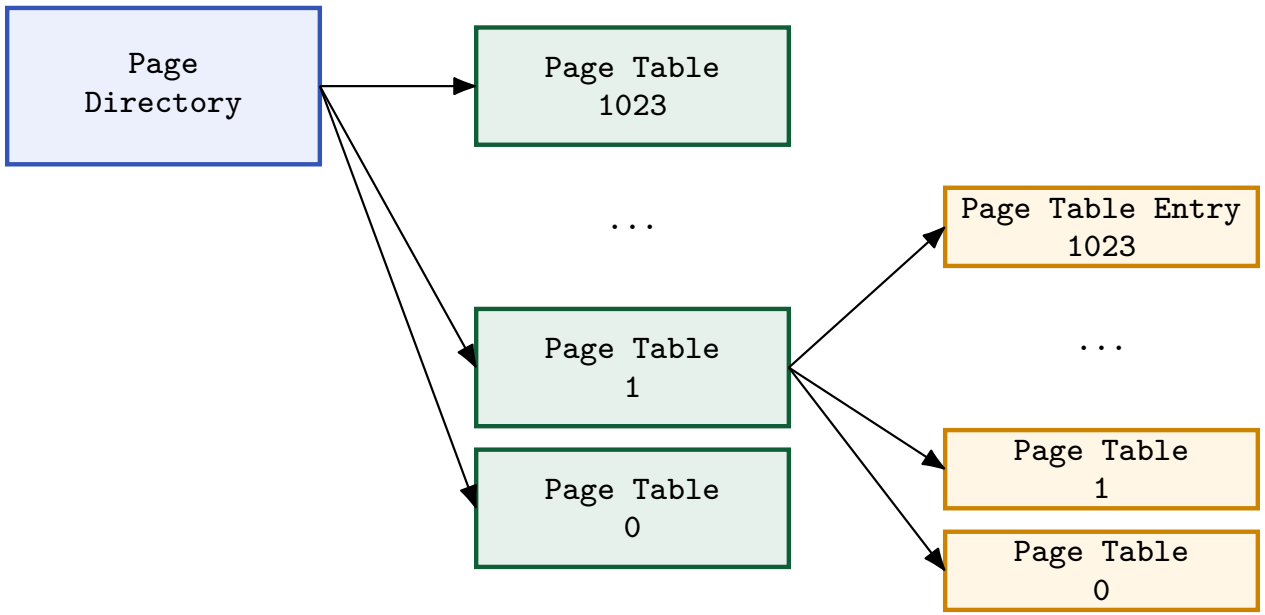


Figure: Relation among Page Directory, Page Tables, and PTEs.

From Virtual to physical

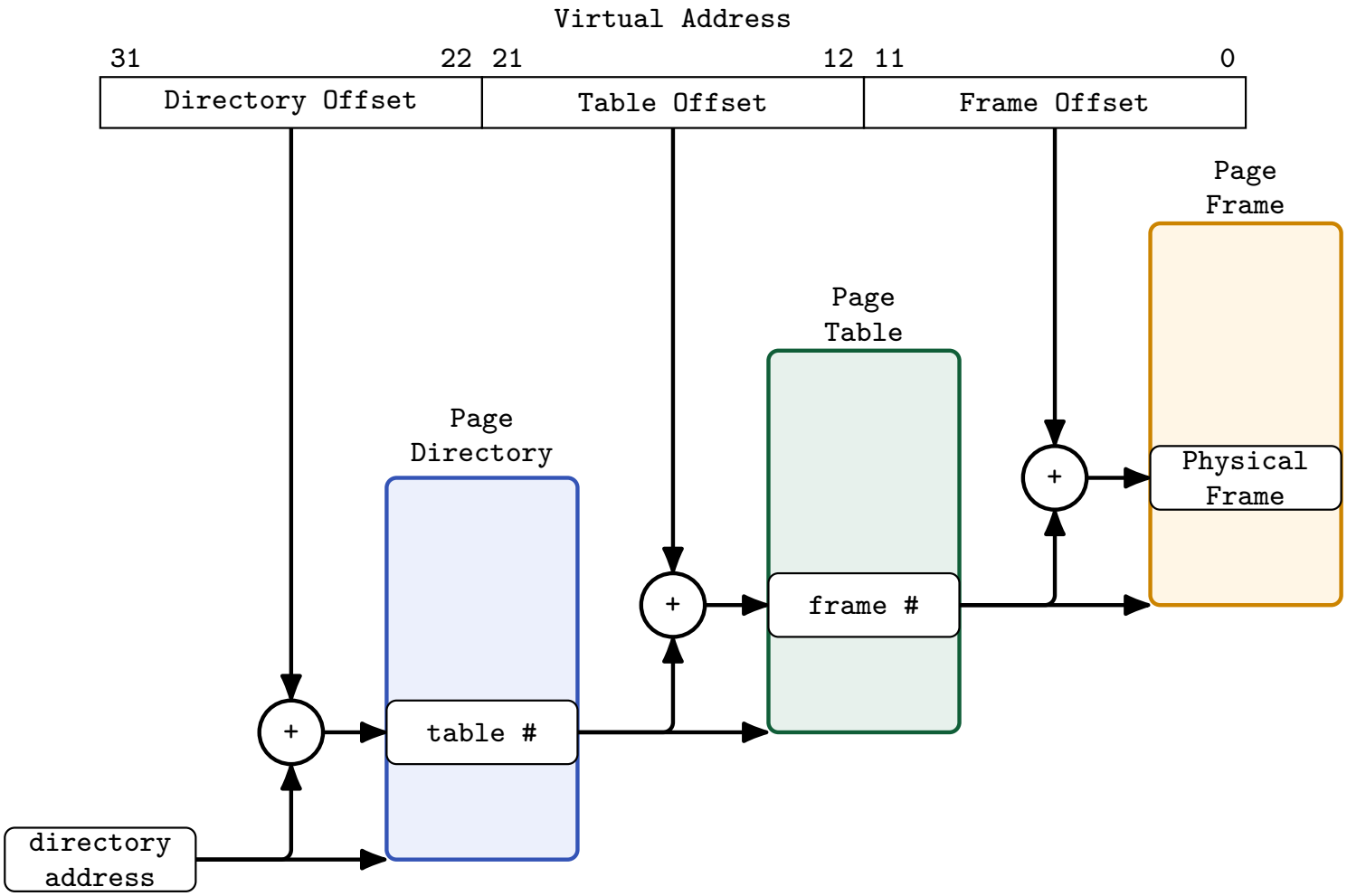


Figure: Generation of physical address from a virtual address (10 bits + 10 bits + 12 bits)

Virtual Memory Management

Memory descriptor

Memory descriptor

The memory descriptor is the Kernel's data structure used to describe:

- **page tables**: The process uses virtual addresses. Page tables let the Memory Management Unit turn a logic address into a physic address.
- **memory regions**: The memory layout of a process is divided into regions (.text, .data, etc), each one having usage permissions and size.

N.B.: Regions of a process are called *segments* in Linux terminology!
Do not mix up the process's regions and memory segmentation.
In the next slides, we will talk about .text, .data regions as segments!

Memory descriptor

The field struct mm_struct **mm** (called memory descriptor) of a task_struct collects the following attributes:

```
struct mm_struct {
    unsigned long start_stack;      // start address of stack segment
    unsigned long mmap_base;        // start address of memory mapping
    unsigned long brk;              // end address of heap segment
    unsigned long start_brk;        // start address of heap segment
    unsigned long end_data;         // end address of data segment
    unsigned long start_data;       // start address of data segment
    unsigned long end_code;         // end address of code segment
    unsigned long start_code;       // start address of code segment
    struct vm_area_struct *mmap;    // list of memory region descr.
    pgd_t * pgd;                    // pointer to page directory
}
```

Memory descriptor

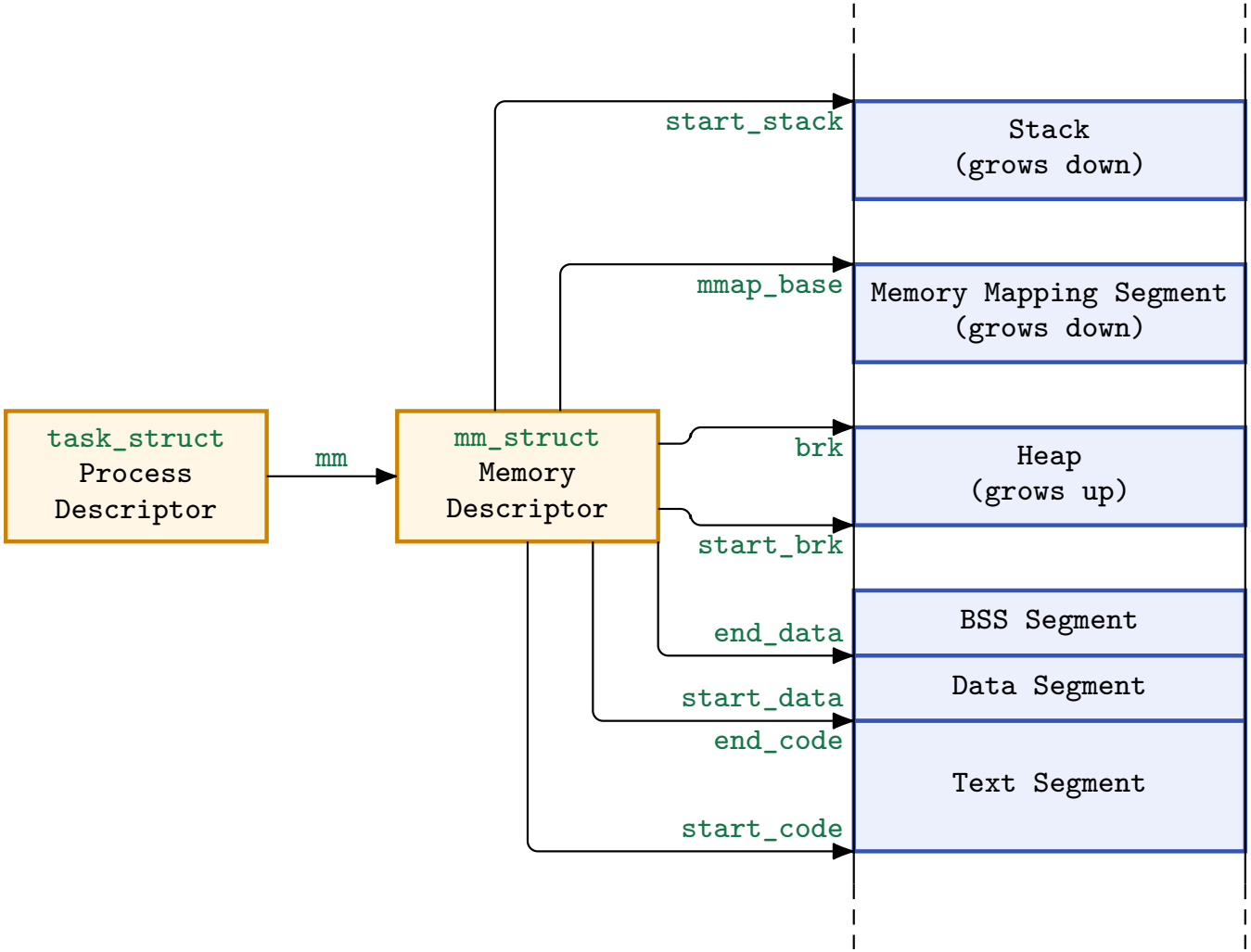


Figure: The process's segments defined by its memory descriptor.

Virtual Memory Management

Segment descriptor

Segment descriptor

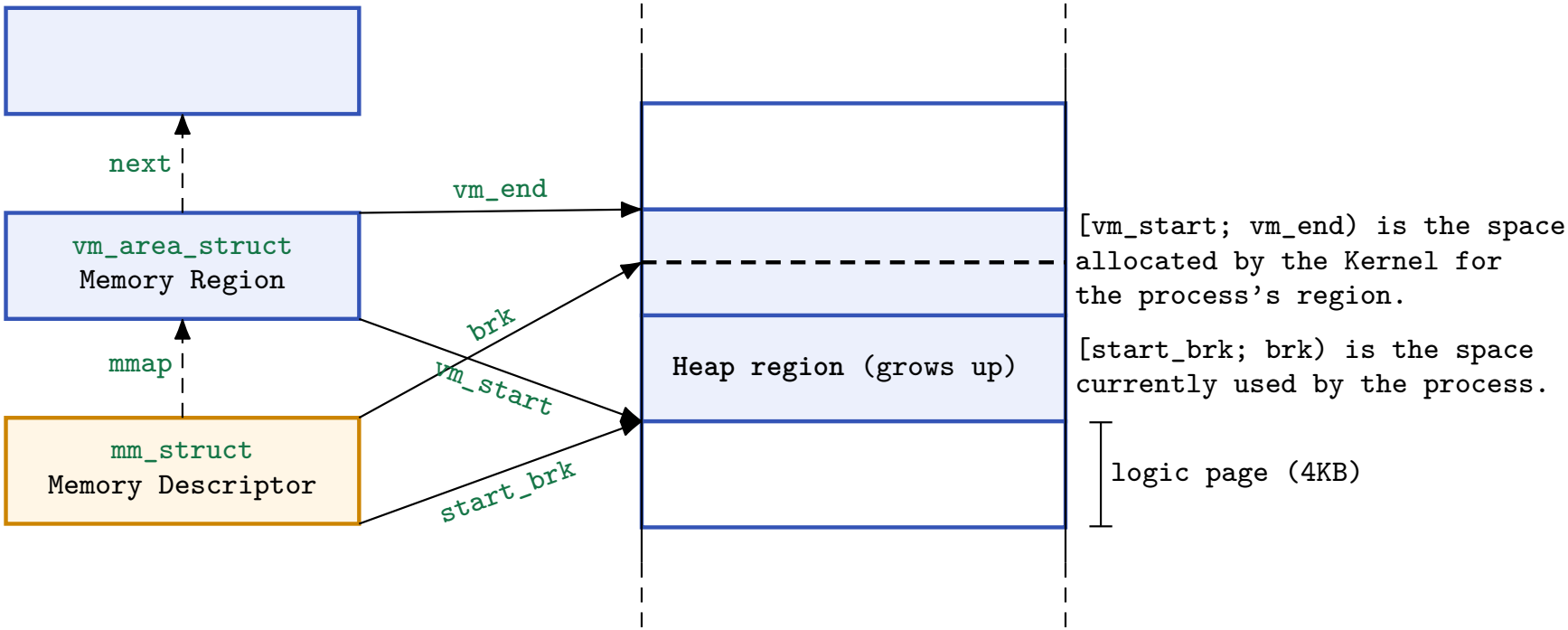
The field vm_area_struct **mmap** of a memory descriptor is the data struct used to represent a contiguous virtual memory area inside a process's segment.

```
struct vm_area_struct {
    unsigned long vm_start;         // start address of segment
    unsigned long vm_end;           // end address of segment
    unsigned long flag;             // access permissions
    struct file *vm_file;           // pointer to mapped file
    struct vm_area_struct *next;  // next region of process
}
```

Why do we have again the start and end address here?
Advice: a page (4KB) is the base unit of memory. When a process ask
for memory, its get back pages from the Kernel.

Segment descriptor

Each virtual memory area identifies a linear address interval of contiguous logic pages, and it has *always* a size multiple to the page's size.

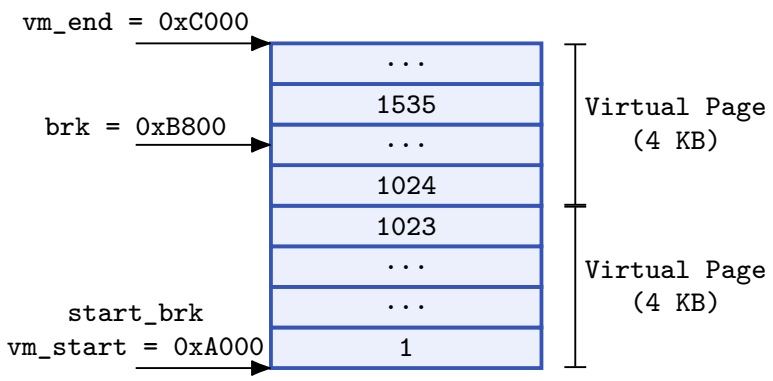


Memory descriptor reports the last byte used inside each process's segment.

Segment descriptor

P.S.: data blocks are here ignored in the heap.

```
size_t size = sizeof(int) * 1536; // 6KB
int* i = (int*) malloc(size);
for (int j = 0; j < 1536; ++j)
    i[j] = j;
```



Heap region of a process allocating 6KB of memory.

- Why did the process get 8KB of memory?
- May the instruction "i[1600] = 0;" cause a segmentation fault?
- If it later requested 1KB of memory, would it get another page?

Segment descriptor

The field **flag** of vm_area_struct struct reports details about all the pages of a process's segment: what they contain, what rights the process has to access each page, how the segment can grow, etc.

Name	Description
VM_READ	Pages can be read
VM_WRITE	Pages can be written
VM_EXEC	Pages can be executed
VM_SHM	The region is used for IPC's shared memory
VM_LOCKED	Pages are locked and cannot be swapped out
VM_GROWSDOWN	The region can expand toward lower addresses
VM_GROWSUP	The region can expand toward higher addresses

Table: A selected list of flags for a segment.

Segment descriptor