OS’25 Project

PART II: **GROUP MODULES**

(DYNAMIC ALLOCATOR, KERNEL HEAP, FAULT HANDLER I**)**

Agenda

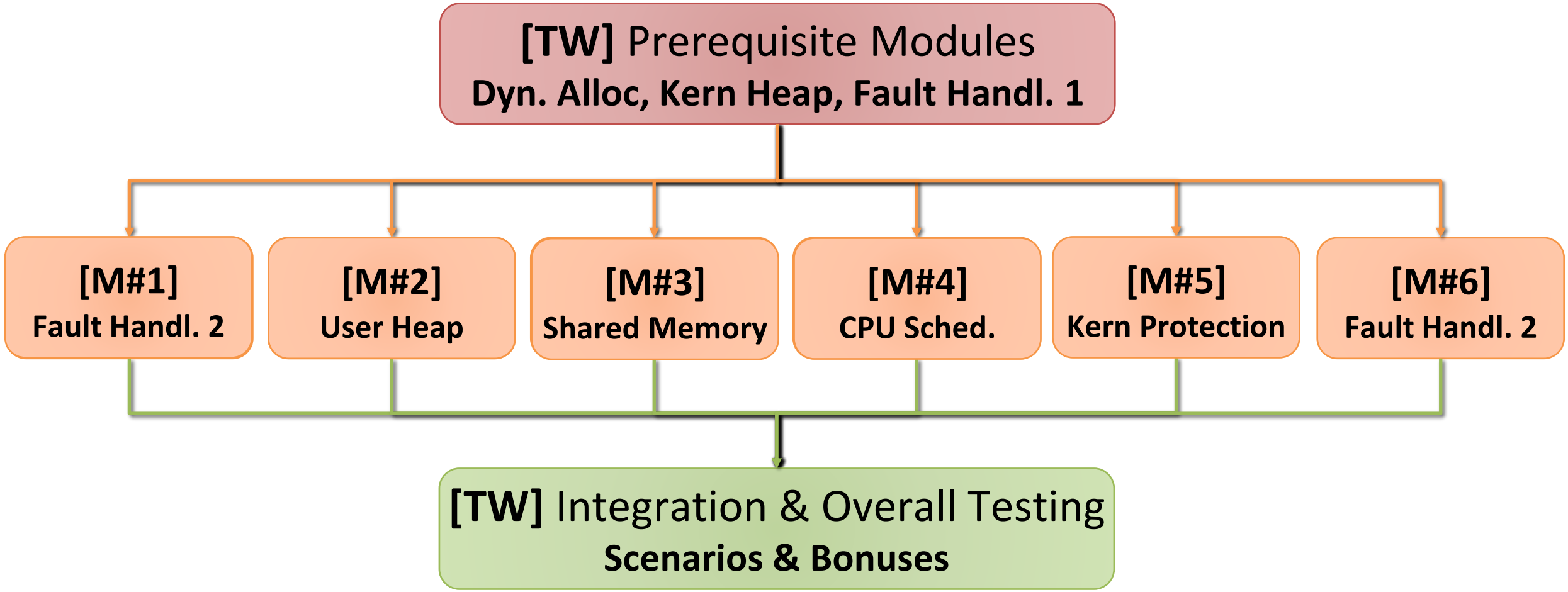
- PART 0: ROADMAP
- PART I: PREREQUISITES
- **PART II: GROUP MODULES**
- **PART III: INDIVIDUAL MODULES**
- **PART IV: OVERALL TESTING & BONUSES**

Agenda

- **PART II: GROUP MODULES**

1. Dynamic Allocator
2. Kernel Heap
3. Fault Handler I (Placement)

Project Overview



Dynamic Allocator

PART II: GROUP MODULES

Objective

Dynamically allocate/free small-size blocks in the heap using **FAST and MEMORY-EFFICIENT way** either for **OS** or any **user program**

Dynamic Allocator – **Functions**

The main functions required to handle "**Dynamic Allocator**" are:

#	Function	File
1	`initialize_dynamic_allocator`	All essential declarations in: inc/dynamic_allocator.h **Functions definitions TO DO: lib/dynamic_allocator.c**
2	`get_block_size`	
3	`alloc_block`	
4	`free_block`	
BONUS#1	`Block If No Free Block`	

Dynamic Allocator – **Idea**

GOAL: allow the dynamic de/allocation of **small-size blocks** in run-time

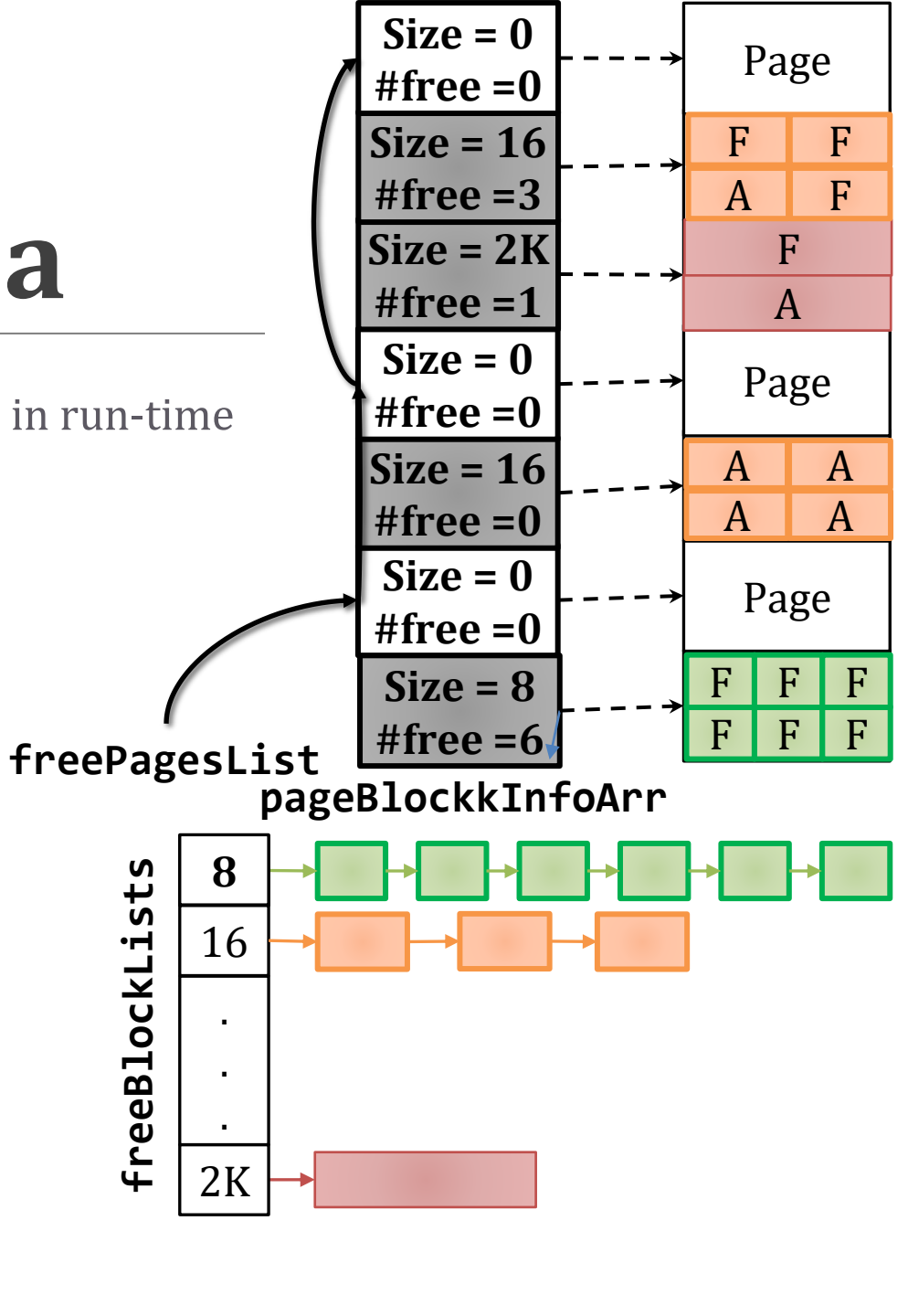
MAIN IDEA:

1. Divide each page into a set of equal-sized blocks
2. Keep track of the free blocks of each size in a linked list
3. For each page in the dynamic allocator space store:
   a. Size of its blocks
   b. num of remaining free blocks
4. Keep track of the free pages in another list

Each list is **double-linked** list if elements, each element has:

prev, next Pointers to the adjacent element

Min block size is **8 B**... **WHY?**

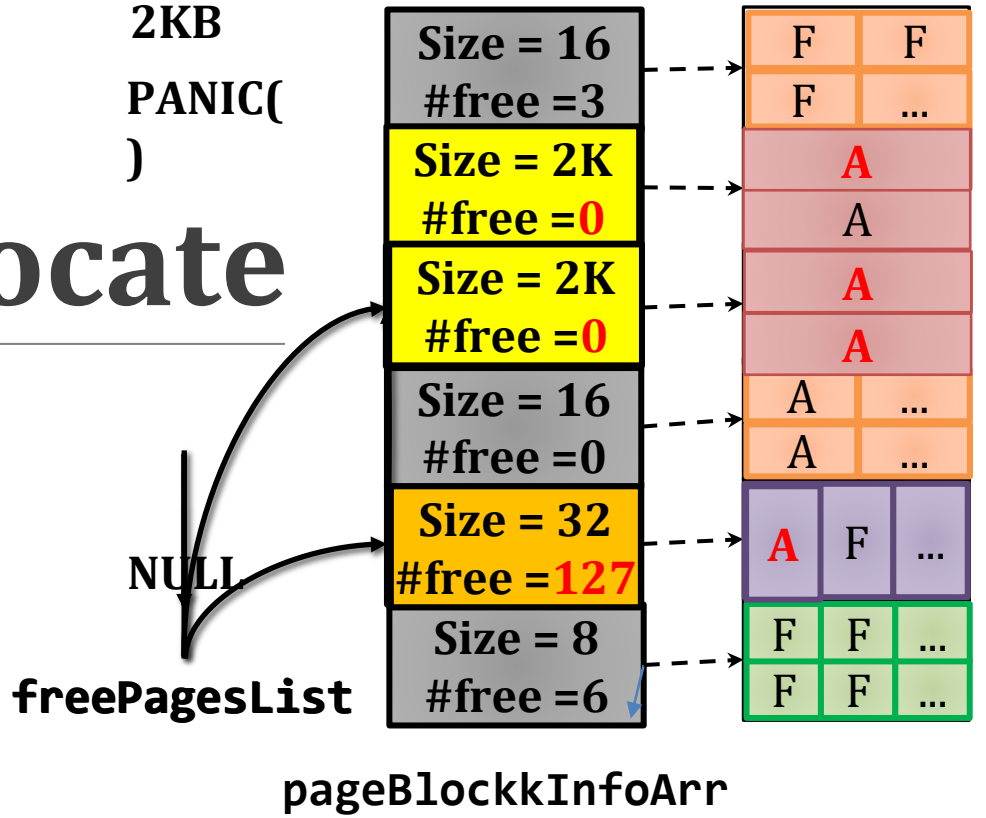


Alloc(1.5 KB) 2KB Alloc(55 B) 2KB

Alloc(20 B) 32B Alloc(62 B) PANIC( )

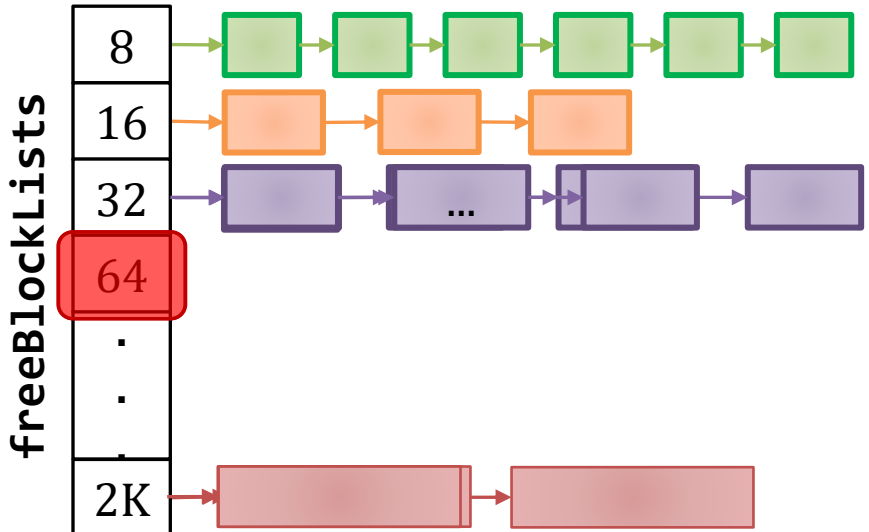
Alloc(2 KB) 2KB

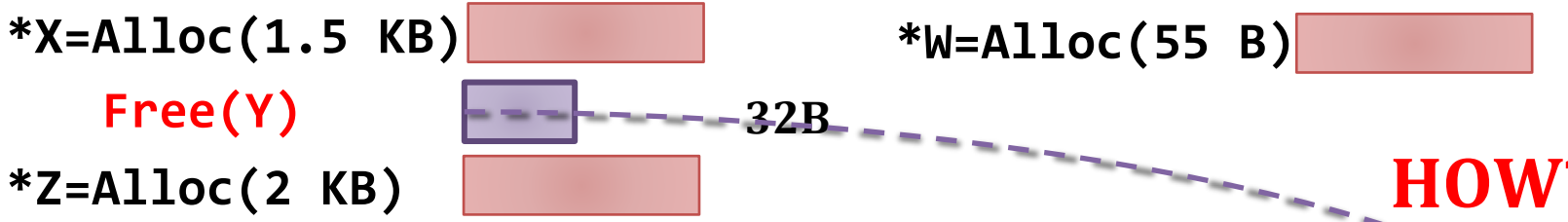
Dynamic Allocator – **Allocate**



- **For** the given size, find its nearest **pow-of-2**

1. **CASE1: if a free block exists**
   - Allocate it & update the data structures
2. **CASE2: else, if a free page exists**
   1. Allocate it from the OS page allocator
   2. Split it into blocks of the desired size
   3. Add these blocks to the corresponding list
   4. Allocate a block & update the data structures
3. **CASE3: else, allocate block from the next list(s)**
4. **CASE4: else, `panic(...)`**



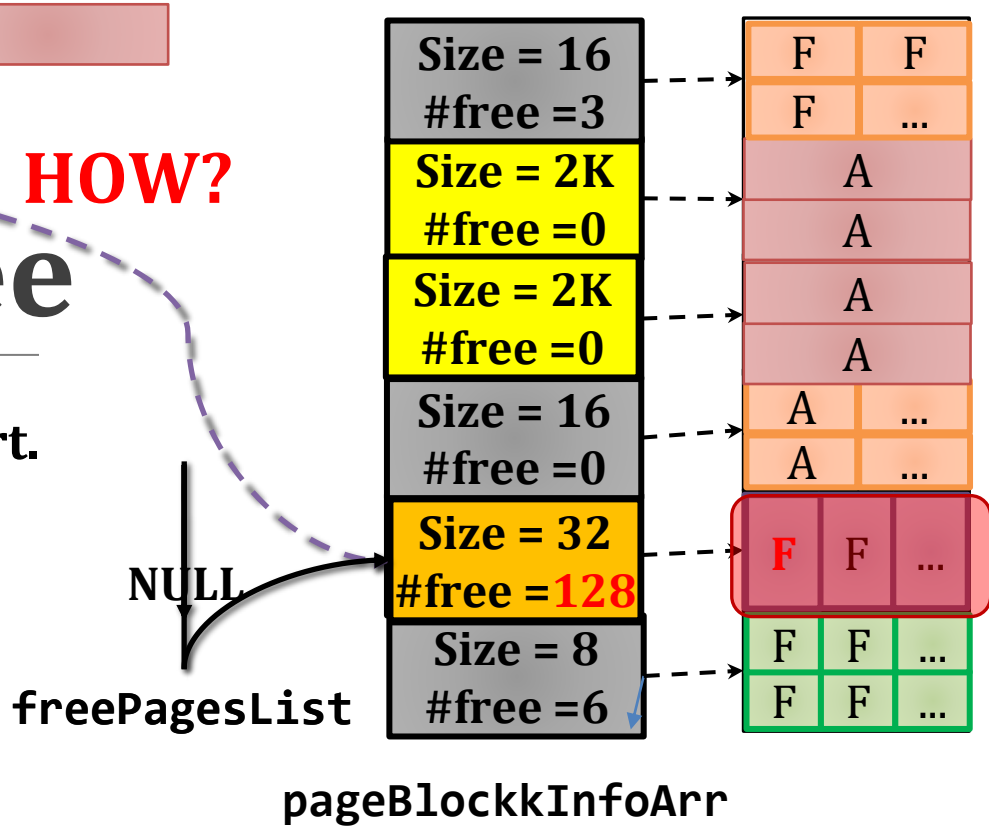


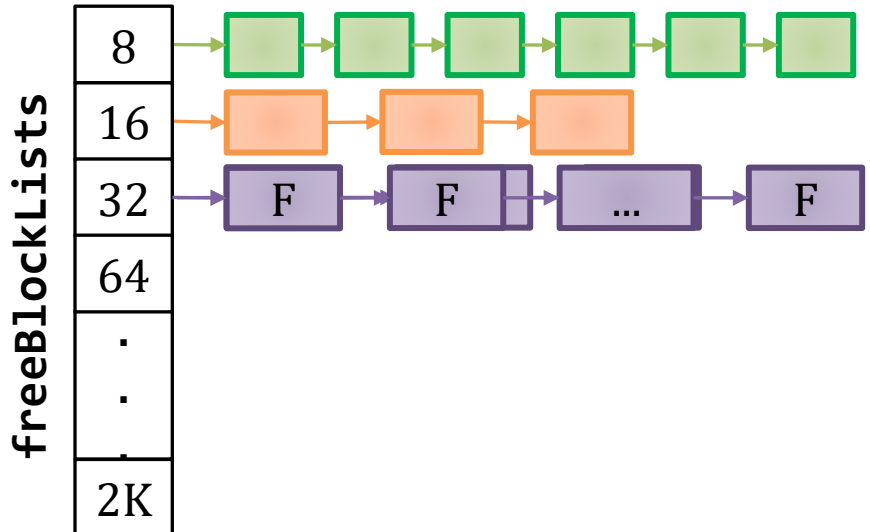
HOW?

Dynamic Allocator – Free

- Free the previously allocated block at the given **Virt. Addr.**

1. **Find corresponding size from `pageBlockkInfoArr`**
2. **Return block to the corresponding list**
3. **Increment #free blocks in `pageBlockkInfoArr`**
4. **If entire page becomes free**
   1. Remove all **its blocks** from corresponding list in `freeBlockLists`
   2. return it to the free frame list
   3. add it to the `freePagesList`





Dynamic Allocator – **Pros & Cons**

PROS:

1. **Allocate** has a **BEST case** of **O(1)** and **BOUNDED** by **O(C)** ; C is constant = max # blocks/page
2. **Free** is **O(1)** and **LINEAR WORST case** of **O(K)** ; K = # free blocks in the corresponding list
3. Can handle **large & small** requests **efficiently**
4. Allocated page can be returned to the page allocator if becomes free can **use** it for **another size**

CONS:

1. **Internal fragmentation** if required size is NOT power-of-two
2. **Bad memory utilization** in case of allocating the request in next level(s)

Dynamic Allocator – **Given Data**

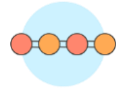
1. Dynamic Allocator Constants

```
#define LOG2_MIN_SIZE (3)                                   //3 Bits
#define LOG2_MAX_SIZE (11)                                  //11 Bits
#define DYN_ALLOC_MAX_SIZE (32<<20)                         //32 MB
#define DYN_ALLOC_MIN_BLOCK_SIZE (1<<LOG2_MIN_SIZE)         //8 BYTE
#define DYN_ALLOC_MAX_BLOCK_SIZE (1<<LOG2_MAX_SIZE)         //2 KB
```

1. Start & End addresses of the Dynamic Allocator inside the corresponding heap

```
//[3] Limits (to be set in initialize_dynamic_allocator())
uint32 dynAllocStart;
uint32 dynAllocEnd;
```

Refer to APPENDICES for:



LISTS Helper Macros

Dynamic Allocator – Given Data

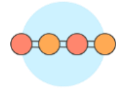
3. Struct containing info about each page with the **prev-next pointer** (to be used as list element)

```
struct PageInfoElement
{
    LIST_ENTRY(PageInfoElement) prev_next_info; /* linked list links */
    uint16 block_size;
    uint16 num_of_free_blocks;
};
```

3. Array of Pages Info & Free Pages List

```
LIST_HEAD(PageInfoElement_List, PageInfoElement);
struct PageInfoElement_List freePagesList ;
struct PageInfoElement pageBlockInfoArr[DYN_ALLOC_MAX_SIZE/PAGE_SIZE];
```

Refer to <u>APPENDICES</u> for:



LISTS Helper Macros

Dynamic Allocator – **Given Data**

5. Block struct containing the **prev-next pointer** (to be used as list element)

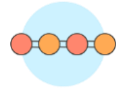
```
struct BlockElement
{
    LIST_ENTRY(BlockElement) prev_next_info;    /* linked list links */
};
```

5. Free Block List of ALL sizes that holds elements of type **`BlockElement`**

```
LIST_HEAD(BlockElement_List, BlockElement);
struct BlockElement_List freeBlockLists[LOG2_MAX_SIZE - LOG2_MIN_SIZE + 1] ;
```

Dynamic Allocator – Given Functions

Refer to APPENDICES for:



LISTS Helper Macros

1. **Get start VA** of the page from the corresponding Page Info pointer

```
uint32 to_page_va(struct PageInfoElement* ptrPageInfo);
```

2. **Get a page** from the Kernel Page Allocator for DA (i.e. Allocate it) (either for **USER** or **KERNEL**)

```
int get_page(void* va);
```

In KERNEL: it calls **alloc_page()** in paging_helpers.c (feel free to edit it if necessarily)

In USER: it calls **__sys_allocate_page**() which, in turn, calls also **alloc_page()**

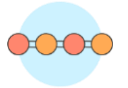
3. **Return a page** to the Kernel Page Allocator (i.e. Free It) (either for **USER** or **KERNEL**)

```
void return_page(void* va);
```

In KERNEL: it calls **unmap_frame()**.

In USER: it calls **__sys_unmap_frame**() which, in turn, calls also **unmap_frame()**

Refer to <u>APPENDICES</u> for:



LISTS Helper Macros

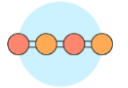
#1: Initialize

```
void initialize_dynamic_allocator(uint32 daStart, uint32 daEnd);
```

Description:

- Initialize the dynamic allocator starting from the given "**daStart**" to "**daEnd**"
- The following items should be initialized here:
  1. **DA Limits**
  2. **Array of Page Info**
  3. **Free Page List**
  4. **Free Block Lists**

Refer to APPENDICES for:



LISTS Helper Macros

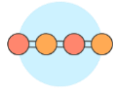
#2: Get Block Size

```
uint32 get_block_size(void *va)
```

Description:

- Get the block size at the given VA from its corresponding entry in the **`pageBlockInfoArr`**

Refer to APPENDICES for:



LISTS Helper Macros

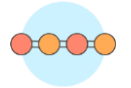
#3: Allocate

```
void *alloc_block(uint32 required_size)
```

Description:

- Allocate new block with the given size
- For the given size, **find** its **nearest** pow-of-two
- **Allocate** new block according to the previously explained cases
- **Return** the start address of the allocated block OR NULL if the requested size is 0

Refer to APPENDICES for:



LISTS Helper Macros

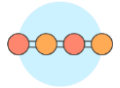
#4: Free

```
void free_block(void* va)
```

Description:

- **Free** the previously allocated block at the given address "**va**"
- **Follow** the previously explained steps

Refer to <u>APPENDICES</u> for:


LISTS Helper Macros

BONUS#1: Block If No Free Block

```
void *alloc_block(uint32 required_size)
```

Description:

- If there's no free block, instead of panic, **BLOCK** the process until a block becomes available

Kernel Heap

PART II: GROUP MODULES

Objective

Allow the kernel to **dynamically** **allocate/free** data

of **different sizes** in run-time

Kernel Heap

IMPORTANT NOTES

Enable the KERNEL HEAP: '**inc/memlayout.h**' set **USE_KHEAP** by **1**

This module is **ESSENTIAL** for all other modules

CAUTION

During your solution, any SHARED data MUST be PROTECTED by critical section via LOCKS

Kernel Heap – **Functions**

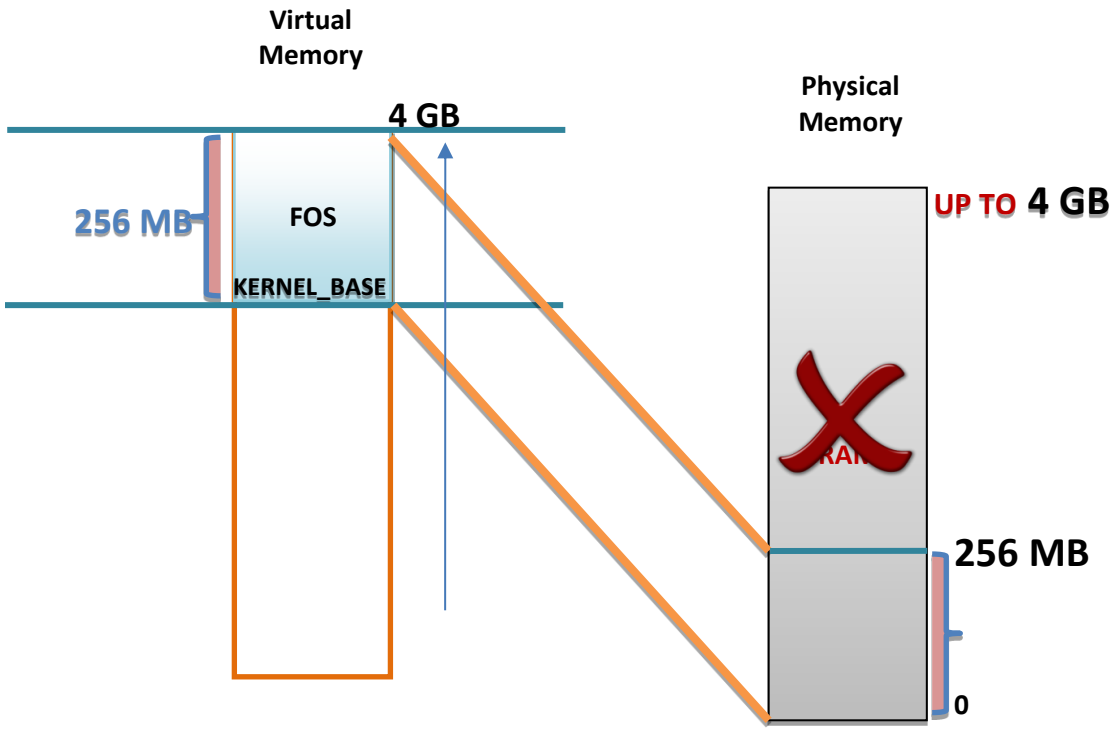
The main functions required to handle "**Kernel Heap**" are:

#	Function	File
1	`kmalloc`	All essential declarations in: Kern/mem/kheap.h **Functions definitions TO DO in: Kern/mem/kheap.c**
2	`kfree`	
3	`kheap_virtual_address`	
4	`kheap_physical_address`	
BONUS 1	`krealloc`	
BONUS 2	`Fast Page Allocator`	

Kernel Heap – What is new?

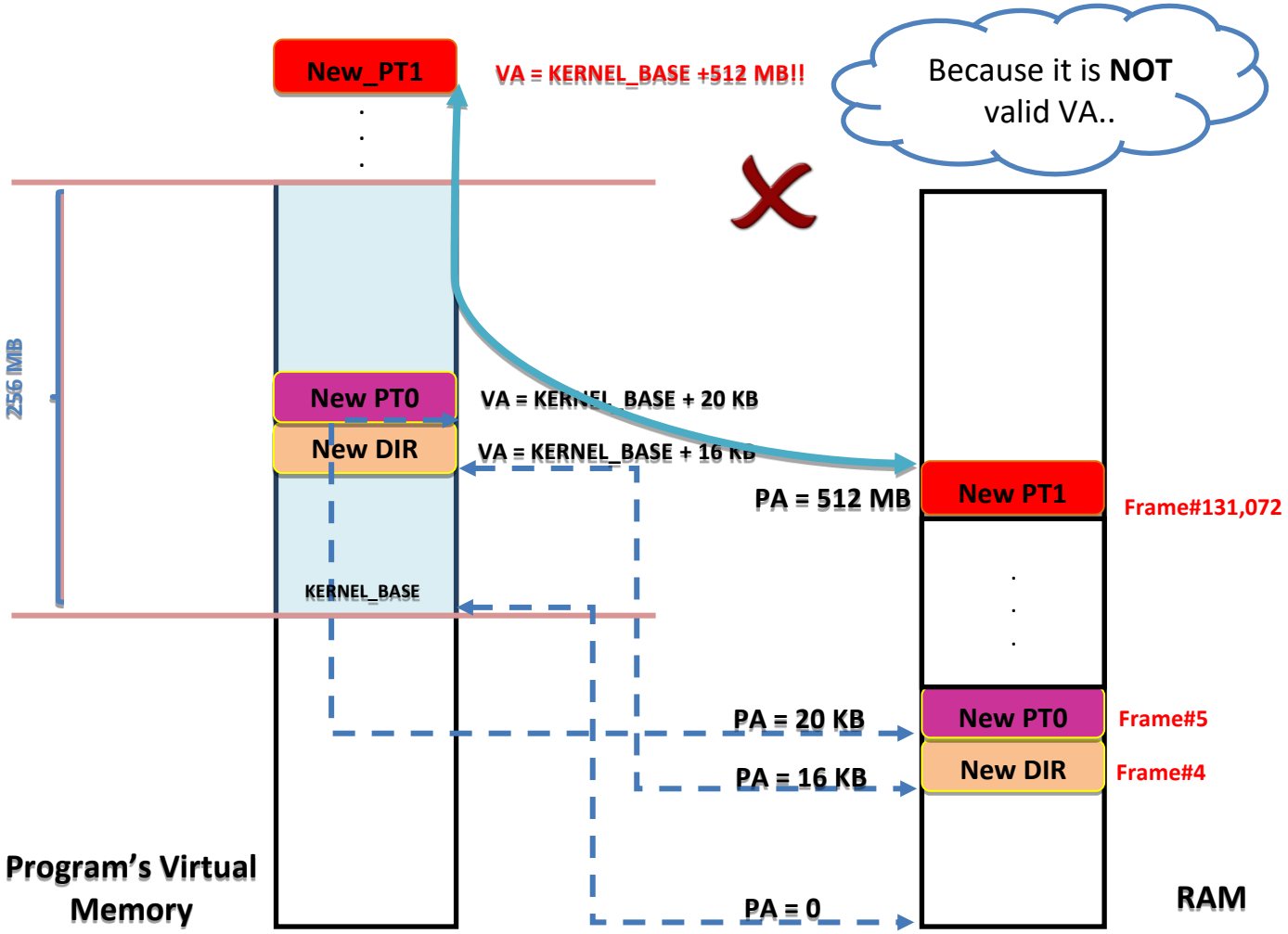
Current: Kernel is **one-to-one** mapped to 256 MB RAM

Problem: Kernel can't directly access beyond 256 MB RAM



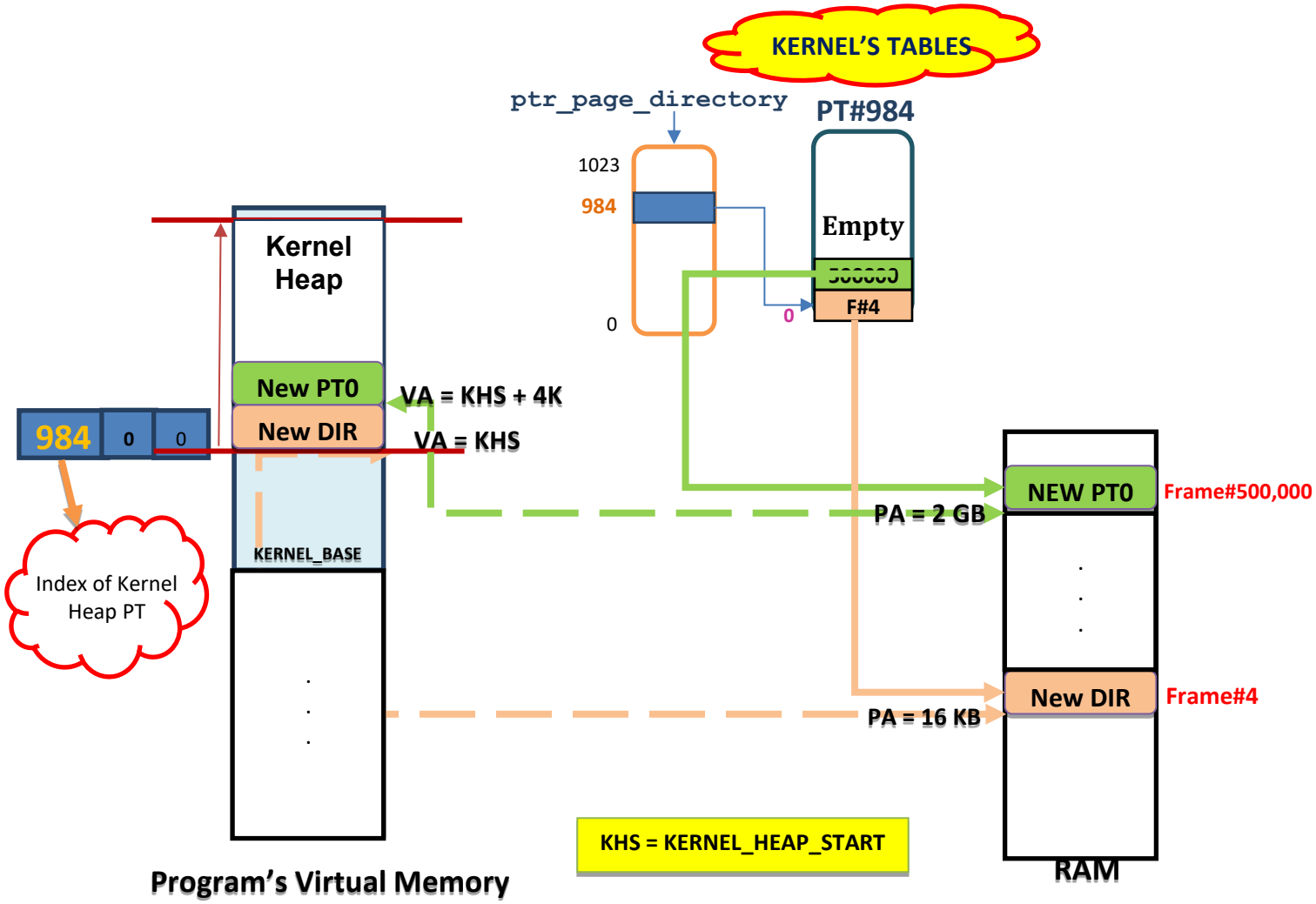
Kernel Heap – What is new?

- **Example:** Kernel can't directly access beyond 256 MB RAM



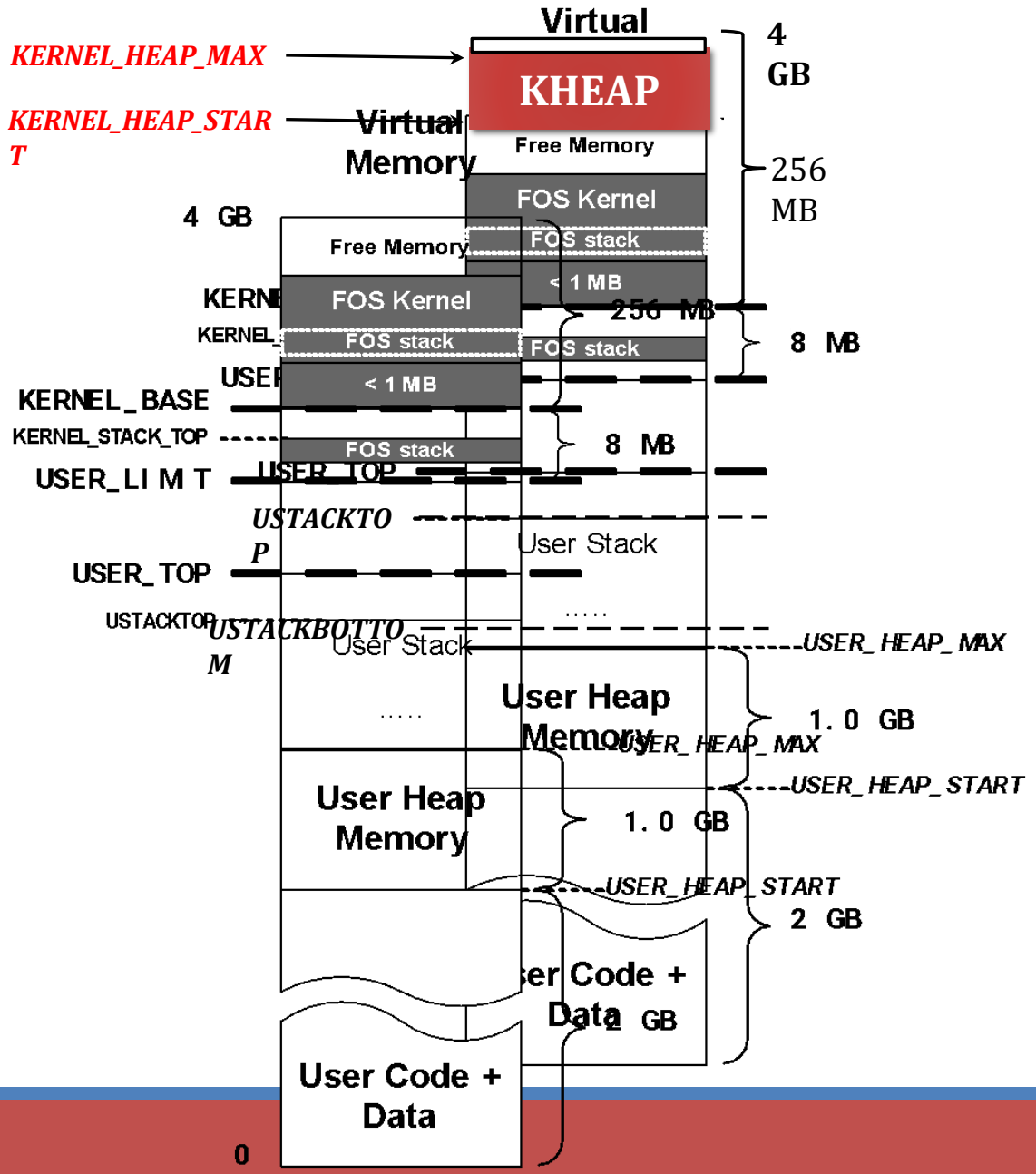
Kernel Heap – What is new?

- Solution: Kernel Heap for dynamic allocations (**No 1-1 map**)



Kernel Heap – What is new?

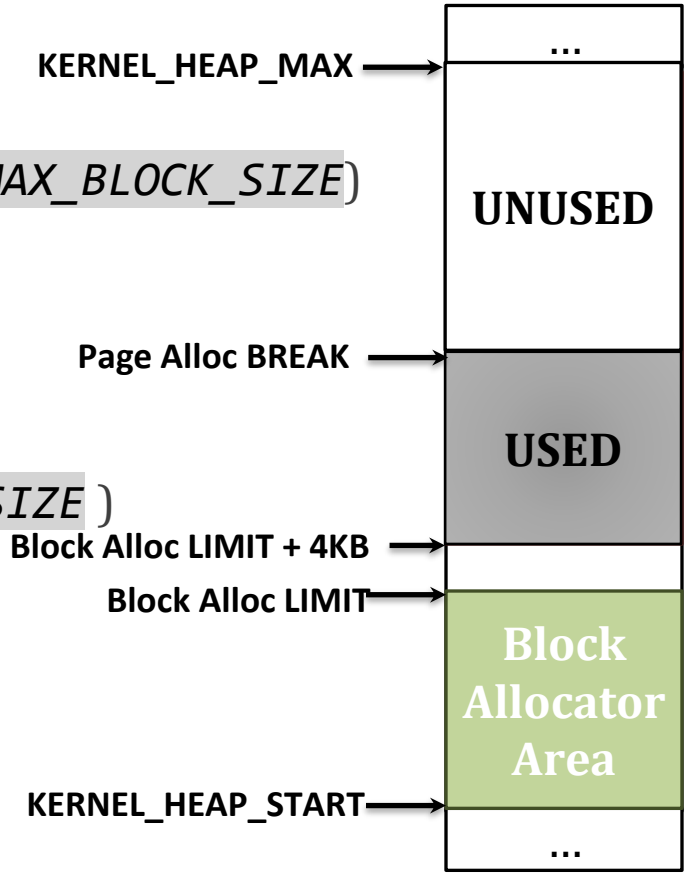
Kernel Heap lies at the end of the virtual space



Kernel Heap – Allocation Types?

There're **TWO** types of allocator

1. **Block Allocator**
   1. Used to allocate **small blocks** (with size **LESS OR EQUAL** *DYN_ALLOC_MAX_BLOCK_SIZE*)
   2. Use Dynamic Allocator Functions
   3. Range: **[**`KERNEL_HEAP_START`, BLK ALLOC LIMIT**)**
2. **Page Allocator**
   1. Used to allocate **chunk of pages** (with size > *DYN_ALLOC_MAX_BLOCK_SIZE*)
   2. Allocation is done on **page boundaries** (i.e. internal fragmentation)
   3. Range: **[**BLK ALLOC LIMIT + `PAGE_SIZE`, `KERNEL_HEAP_MAX`**)**
      1. **USED** Area: **[**BLK ALLOC LIMIT + `PAGE_SIZE`, PAGE ALLOC BREAK**)**
      2. **UNUSED** Area: **[**PAGE ALLOC BREAK , `KERNEL_HEAP_MAX`**)**

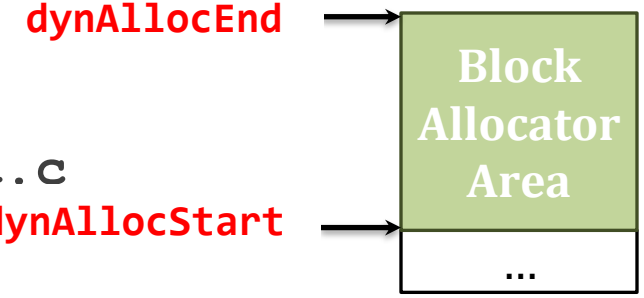


Kernel Heap – **Block Allocator**

1. Has 2 limits:
   1. `dynAllocStart`: begin of block allocator area
   2. `dynAllocEnd`: end of block allocator area
2. Use Dynamic Allocator with its data structure
3. **Already initialized**, together with the dynamic allocator itself inside:

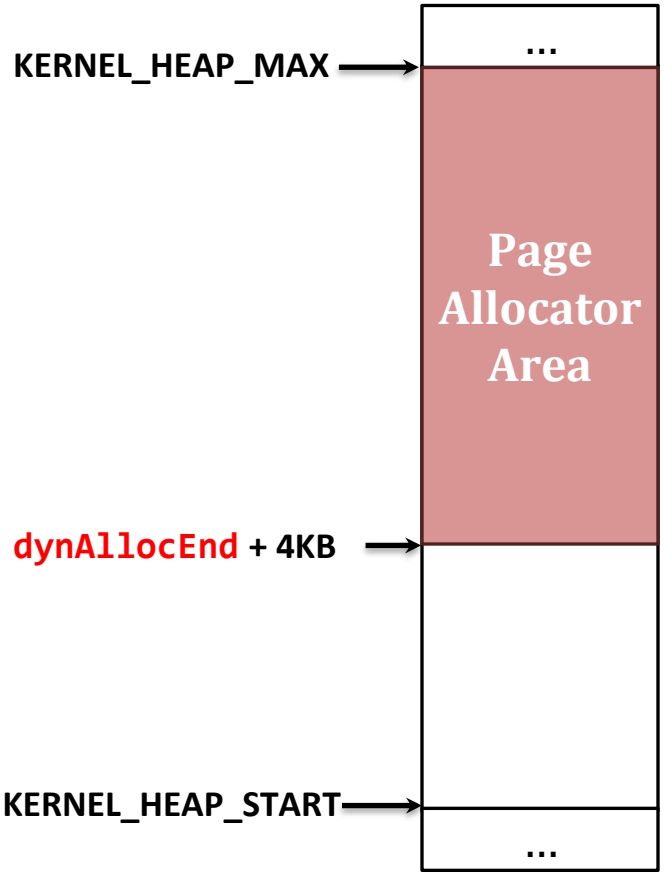
`int` **`kheap_init(…)`** defined in **`kern/mem/kheap.c`**

- This function, in turn, is **already called** in **`FOS_initialize`**() in **`init.c`**



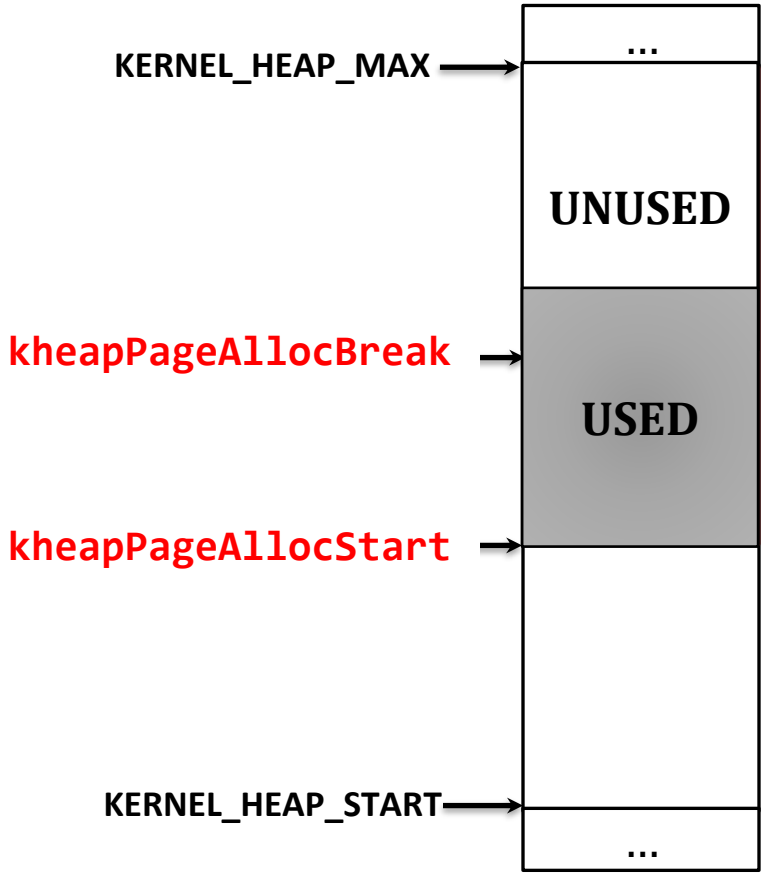
Kernel Heap – **Page Allocator**

- Should start at **one-page after** the **block allocator** limit
- Allocation is done on **page boundaries (**multiple of 4KB**)**
  - i.e. **internal fragmentation** can occur
- All required pages should be **allocated** & **mapped** by OS
- Allocation Strategy: **CUSTOM FIT**



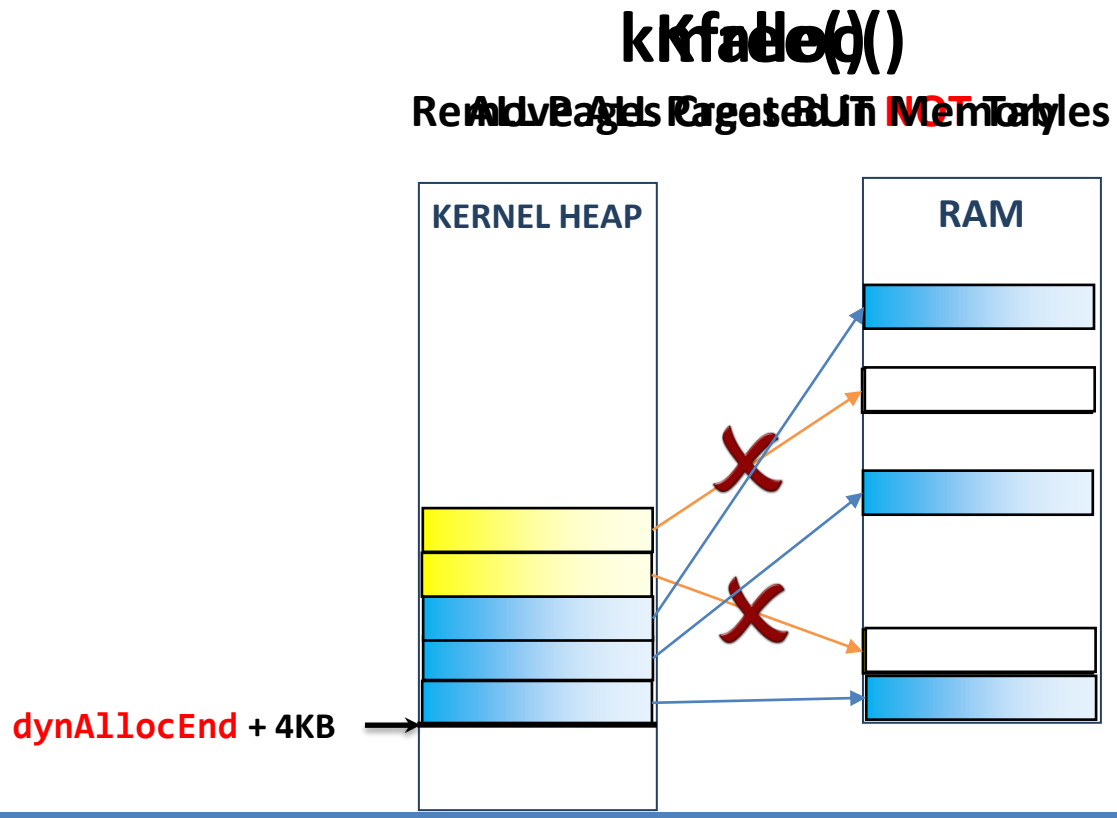
Kernel Heap – **Page Allocator**

1. Has 2 limits defined in **`kern/mem/kheap.h`**:
    1. `kheapPageAllocStart`: begin of page allocator area
    2. `kheapPageAllocBreak`: end of currently used area
2. malloc/free can move `kheapPageAllocBreak` up/down



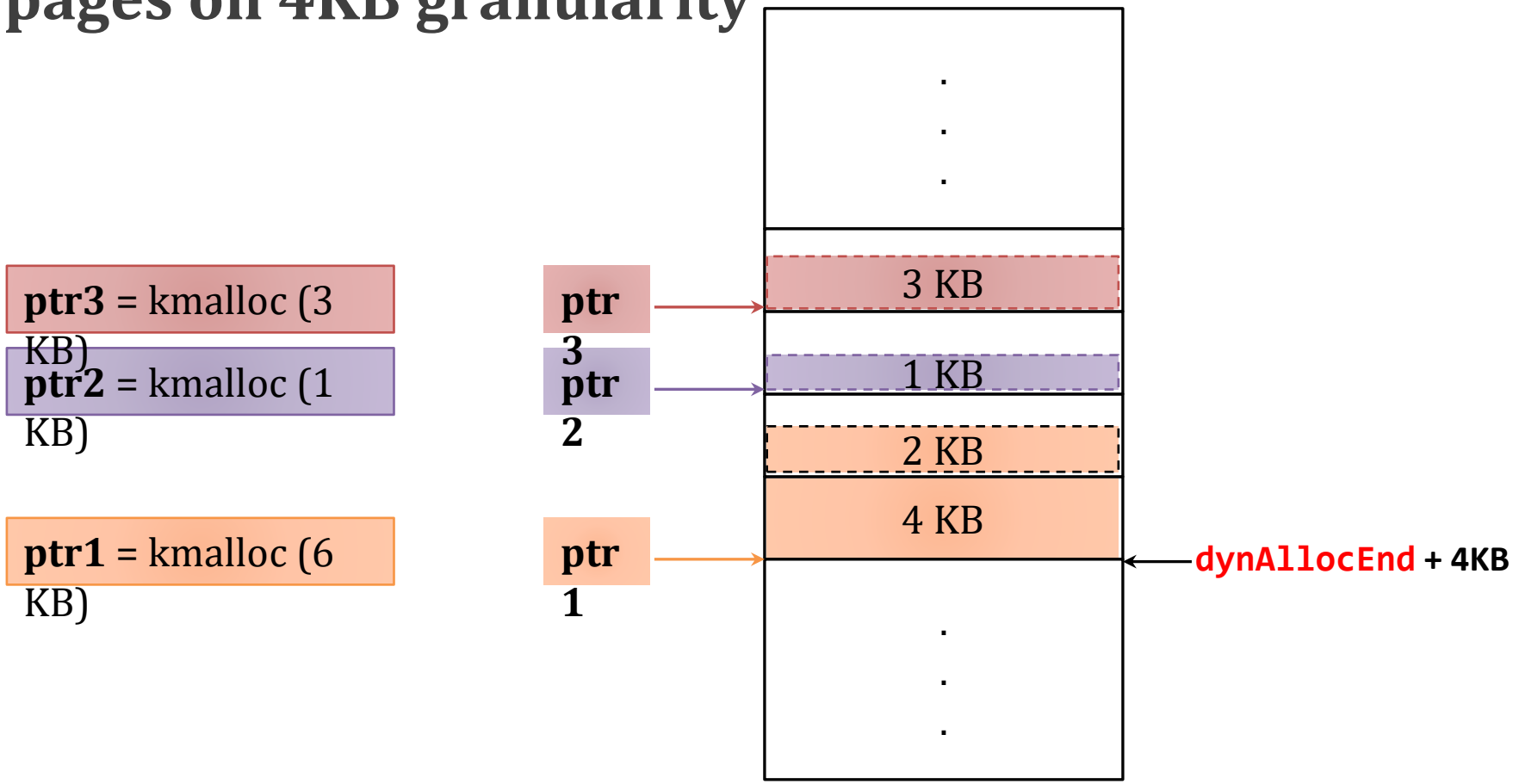
Kernel Heap – **Page Allocator**

1. **kmalloc():** dynamically allocate space
2. **kfree():** delete a previously allocated space



Kernel Heap – Page Allocator (kmalloc)

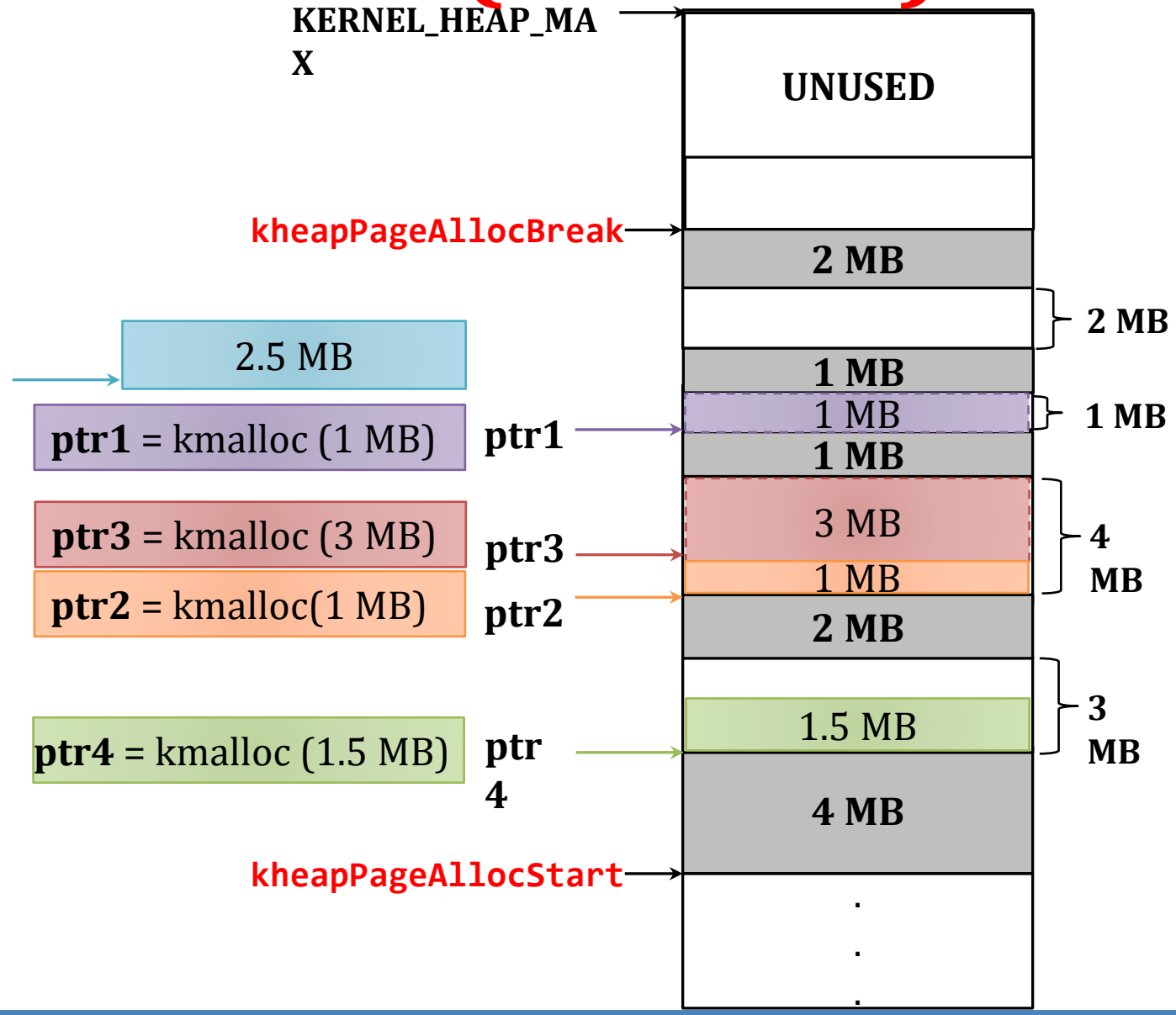
Allocate pages on 4KB granularity



Kernel Heap – Page Allocator (kmalloc)

CUSTOM FIT Strategy

1. Search for **EXACT fit**
2. if not found,
   search for **WORST fit** till break
3. if not found,
   extend **BREAK** if available
4. if not available,
   return **NULL**

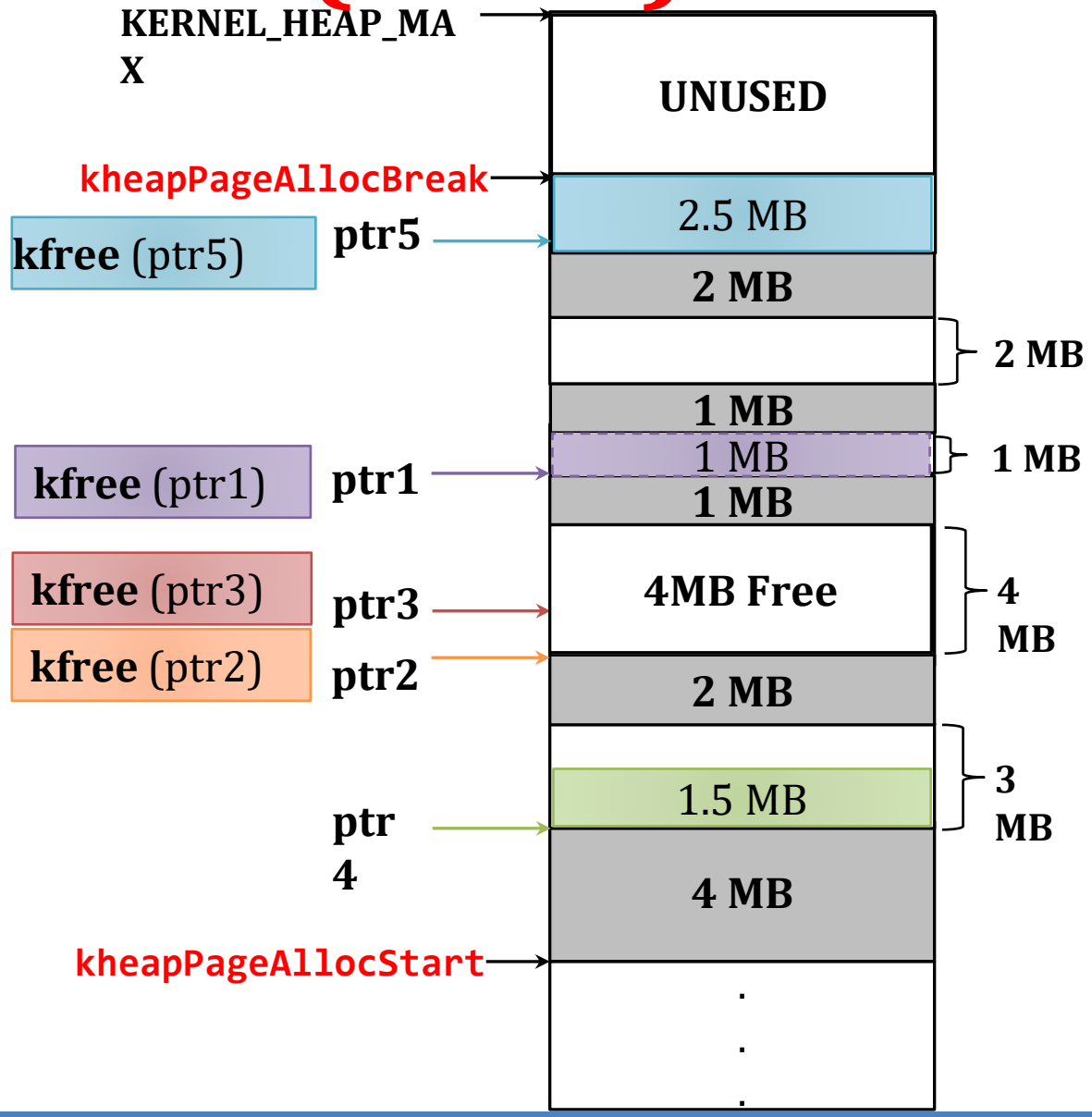


Kernel Heap – Page Allocator (kfree)

Make sure to:

1. Merge adjacent blocks

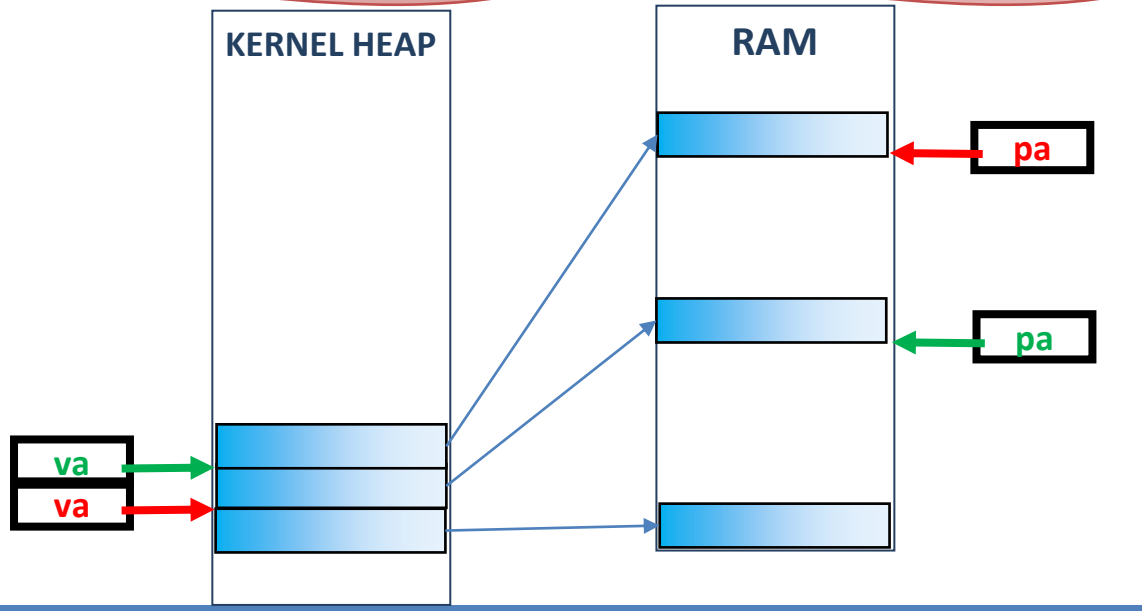
2. Update the **`kheapPageAllocBreak`** if freeing the last space



Kernel Heap – **Page Allocator**

3. **kheap_physical_address():** find physical address of the given kernel one
4. **kheap_virtual_address():** find kernel virtual addr of the given physical one

IMP: Both MUST be Efficient ~O(1)



#5: kmalloc()

```
void* kmalloc(unsigned int size)
```

Description:

1. If size ≤ *DYN_ALLOC_MAX_BLOCK_SIZE*: **[BLOCK ALLOCATOR]**
   - Use dynamic allocator to allocate the required space
2. Else: **[PAGE ALLOCATOR]**
   - Allocate & map the required space on page-boundaries using CUSTOM FIT strategy

- If failed to allocate: return NULL

#6: kfree()

```
void kfree(void* virtual_address)
```

Description:

1. If virtual address inside the **[BLOCK ALLOCATOR]** range
   - Use dynamic allocator to free the given address
2. If virtual address inside the **[PAGE ALLOCATOR]** range
   - FREE the space of the given address from RAM
3. Else (i.e. invalid address): should **`panic(...)`**

#7: kheap_physical_address()

```
unsigned int kheap_physical_address(unsigned int virtual_address)
```

Description:

1. return the physical address corresponding to given virtual_address (**including offset**)
2. If no mapping, return 0.
3. It should work for both **[BLOCK ALLOCATOR]** and **[PAGE ALLOCATOR]**
4. It should run in **O(1)**

#8: kheap_virtual_address()

```
unsigned int kheap_virtual_address(unsigned int physical_address)
```

Description:

1. return the virtual address corresponding to given physical_address (**including offset**)
2. If no mapping, return 0.
3. It should work for both **[BLOCK ALLOCATOR]** and **[PAGE ALLOCATOR]**
4. It should run in **O(1)**

BONUS#1: krealloc()

```
void *krealloc(void *virtual_address, uint32 new_size)
```

Description:

1. Attempts to resize the allocated space at given virtual address to "**new size**" bytes, possibly moving it in the heap.
2. If **successful**, returns the **new virtual address**.
3. On **failure**, returns a **NULL** pointer, and the old virtual address remains valid.
4. A call with virtual_address = null is equivalent to kmalloc()
5. A call with new_size = zero is equivalent to kfree()
6. It should work in **[BLOCK ALLOCATOR] , [PAGE ALLOCATOR]** and **CROSS-ALLOCATORS**

BONUS#2: Fast Page Allocator

Description:

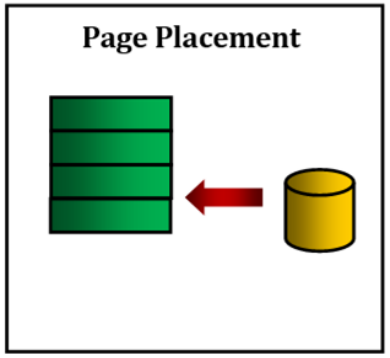
EFFICIENT implementation of the **Page Allocator** using **suitable data structures**

Fault Handler I
(Placement)

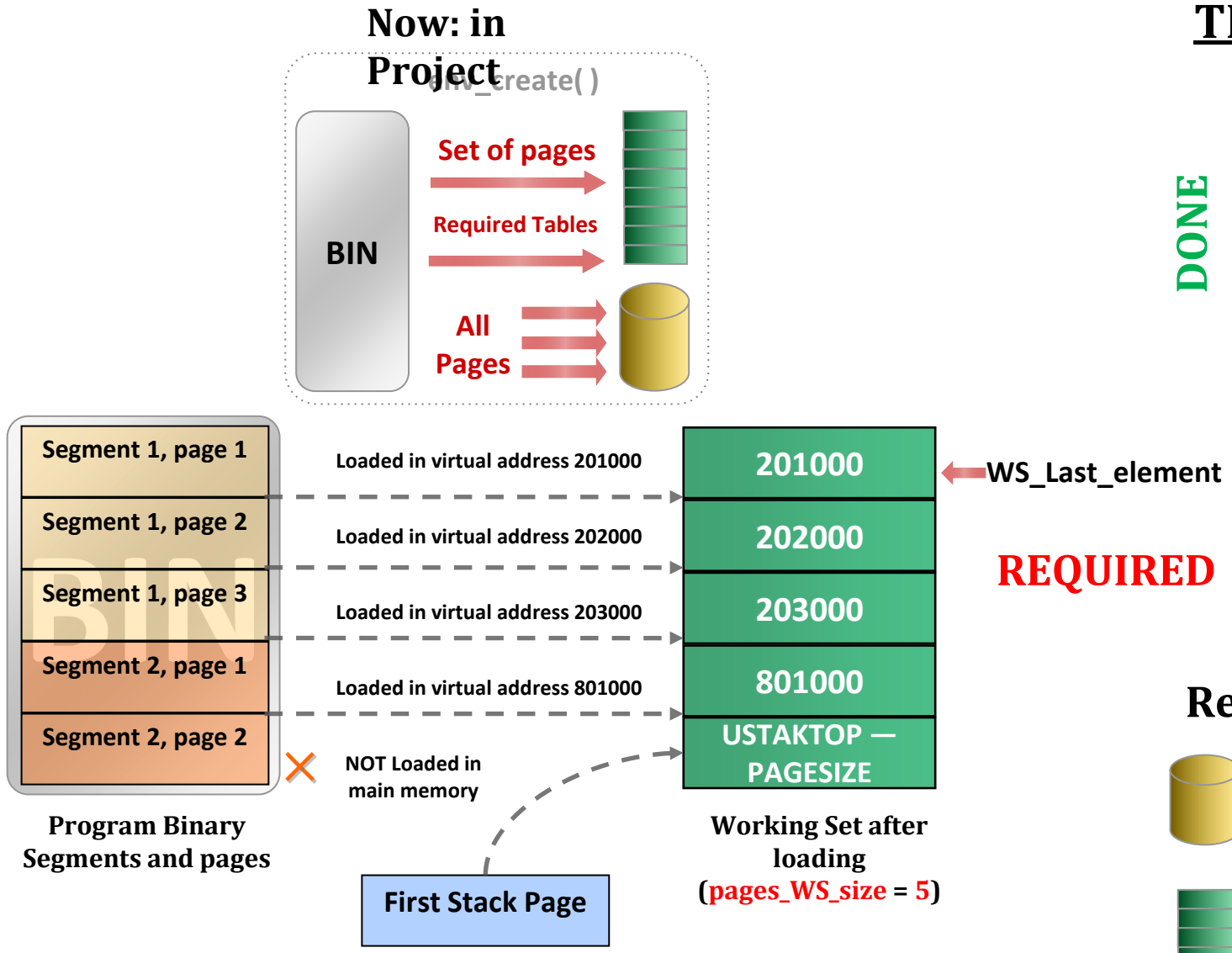
PART I: PREREQUISITES

Objective

Handle the page fault exception by retrieving the faulted page **from H.D.D to memory**



Load Program [env_create]



THREE kernel dynamic allocations:

DONE

1. **`create_page_table()`:** create new page table and link it to directory.
2. **`create_user_directory()`:** create new user directory.
3. **`create_user_kern_stack(...)`:** create new user kernel stack.

REQUIRED

Refer to APPENDICES for:

- Page File Helper Functions
- Working Set Structure & Helper Functions

Working Set: **Structure**

inc/environment_definitions.h

```
struct Env {
    //...
    //================
    /*WORKING SET*/
    //================
    //page working set management
    struct WS_List page_WS_list ;   FIFO, CLK, ModCLK //List of WS elements
    struct WorkingSetElement* page_last_WS_element; //ptr to last inserted WS element
    unsigned int page_WS_max_size;                  //Max allowed size of WS
```

Each Element

Proc Limit

inc/environment_definitions.h

```
struct WorkingSetElement {
    unsigned int virtual_address;
    unsigned int time_stamp ;  LRU
    unsigned int sweeps_counter;
    LIST_ENTRY(WorkingSetElement) prev_next_info;
```

- Each process has a **working set LIST** that is initialized in **`env_create`**()
- Its **max size** is set in "**`page_WS_max_size`**" during the **`env_create`**()
- "**`page_last_WS_element`**" will point to either:
  - the **next location** in the WS after the last set one If **list is full**.
  - **Null** if the list is **not full**.
- This list hold pointers to **`struct`** containing info about the currently loaded pages in memory.
- Each struct holds two important values about each page:
  1. User virtual address of the page
  2. Previous & Next pointers to be used by list

Working Set: **Functions [GIVEN]**

```
struct WorkingSetElement* env_page_ws_list_create_element
        (struct Env* e, uint32 virtual_address)
```

Description:

1. **Allocate** a new object of `struct WorkingSetElement` (using `kmalloc()`)
2. **Initialize** it by the given virtual address

Return

1. On success: pointer to the created object
2. On failure: kernel should `panic()`

Working Set: **Functions [GIVEN]**

```
void env_page_ws_print(struct Env* e)
```

Description:

- Print the page working set **virtual addresses** together with **used**, **modified** & **buffered** bits.
- It also shows where the **page_last_WS_element** of the working set is point to

Working Set: **Functions [GIVEN]**

```
void env_page_ws_invalidate(struct Env* e, uint32
                            virtual_address)
```

Description:

- Search for the given virtual address inside the working set of "**e**" **,** if found:

1. **Remove** its WS element from the **list**
2. **Delete** this element from the kernel **memory** (using **`kfree()`**)
3. **Unmap** the page from the address space

#9: Kernel Dyn. Alloc. for a Process

```
void* create_user_kern_stack(uint32* ptr_user_page_directory)
```

Description:

1. **Create** a user kernel stack of size ***KERNEL_STACK_SIZE***
2. **Mark** its **bottom** page as **NOT PRESENT** (GUARD page)

Return

1. On success: pointer to the created stack
2. On failure: kernel should **`panic()`**

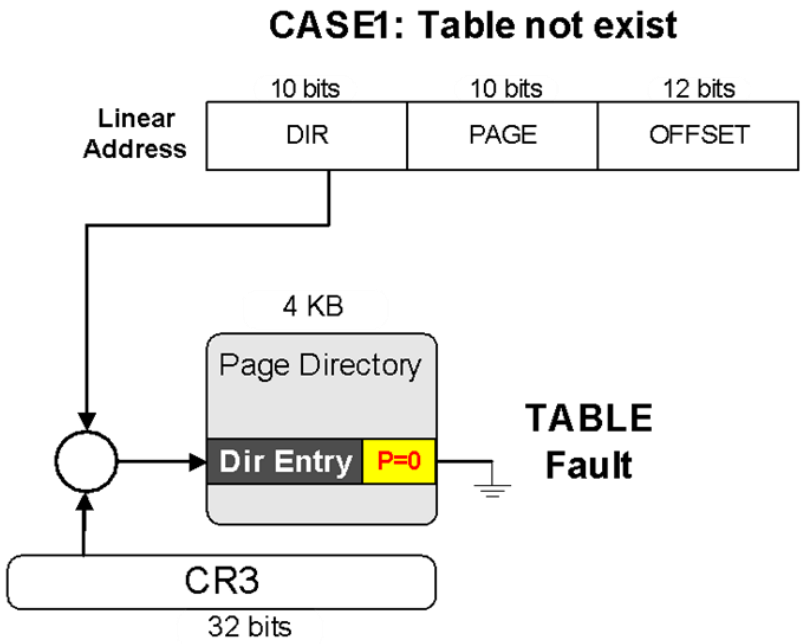
Fault Handler I: **Overview**

The main functions required to handle "**Page Fault**" are:

#	Function	File
1	`fault_handler`	**Functions definitions TO DO in: kern/trap/fault_handler.c**
2	`page_fault_handler`	

Fault Handler I: **Overview**

- **Fault:** is an exception thrown by the processor (MMU) to indicate that:
  - **A page table** not exist in the main memory (i.e. new table). (see the following figure) **OR**
  - **A page** can't be accessed due to either it's not present in the main memory



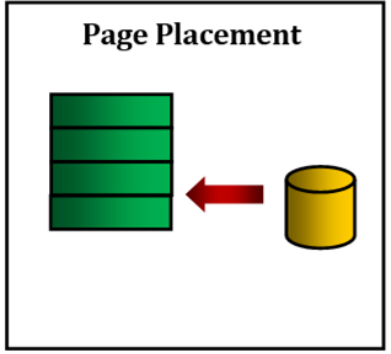
#10: Check Invalid Pointers

```
void fault_handler(struct Trapframe *tf)
```

Description:

- **Check** the **`faulted_va`** on the following**:**
  1. Pointing to **UNMARKED** page in user heap (i.e. its **`PERM_UHPAGE`** should equals 0)
  2. Pointing to **kernel**
  3. **Exist** with **read-only** permissions
- If **invalid (**any of above occur**)**: it must be rejected without harm to the kernel or other running processes, by **exiting** the process using **env_exit()**

#11: Placement



Refer to <u>APPENDICES</u> for:

Page File Helper Functions

Working Set Struct & Helper Func's

```
page_fault_handler(struct Env * falulted_env, uint32 fault_va)
```

If the size of the page working LIST < ***its max size,*** then do (refer to appendices for helper functions)

Scenario 1: Placement

1. **Allocate** space for the faulted page
2. **Read** the faulted page from page file to memory
3. If the page **does not exist** on page file, then
    1. If it is a **stack** or a **heap** page, then, it's OK.
    2. Else, it must be **rejected** without harm to the kernel or other running processes, by **exiting** the process.
4. Reflect the changes in the page working set list (i.e. add new element to list & update its last one)

NEXT, IT'S **YOUR TURN…**

☺ Enjoy **developing** your **own OS** ☺