OS’24 Project

MILESTONE 2: **MEMORY**

KERNEL HEAP, USER HEAP, SHARING & FAULT HANDLER I

Agenda

- Logistics
- Part 0: Code Updates
- Part 1: Kernel Heap
  - Block Allocator
  - Page Allocator
- Part 2: Fault Handler I
- Part 3: User Heap
  - Block Allocator
  - Page Allocator
- Part 4: Shared Memory
- Summary & Quick Guide
- How to submit?

Logistics

Dependency:

- MS1: dynamic allocator (`alloc_block_FF` & `free_block`)

Delivery Method: GOOGLE FORMS

- It's **FINAL** delivery
- **MUST** deliver the required tasks and **ENSURE** they're worked correctly

Delivery Dates:

- **THU of Week #9 (28/11 @11:59 PM)**
- Upload your code **EARLY** as **NO EXCEPTION** will be accepted.

Support:

- The support for teams will be through their **MENTORS ONLY (+Lecturer)** during via:

1. MAIN METHOD: weekly office hours.
2. SECONDARY METHOD [OPTIONAL]: other contact method [**MUST declare your Team# first**]

Logistics

ADVICE#1: WORK AS A TEAM

Milestone 2: MEMORY

1. Kernel Heap: 6 functions
2. Fault Handler I: 3 functions
3. User Heap: 6 functions
4. Shared Mem: 6 functions

MUST be finished FIRST
Expected: Before MT

≈ **3~4 Functions/member**
on **3 Weeks**

L1 □ 9 FUNCTIONS - L2 □ 10 FUNCTIONS - L3 □ 2 FUNCTIONS

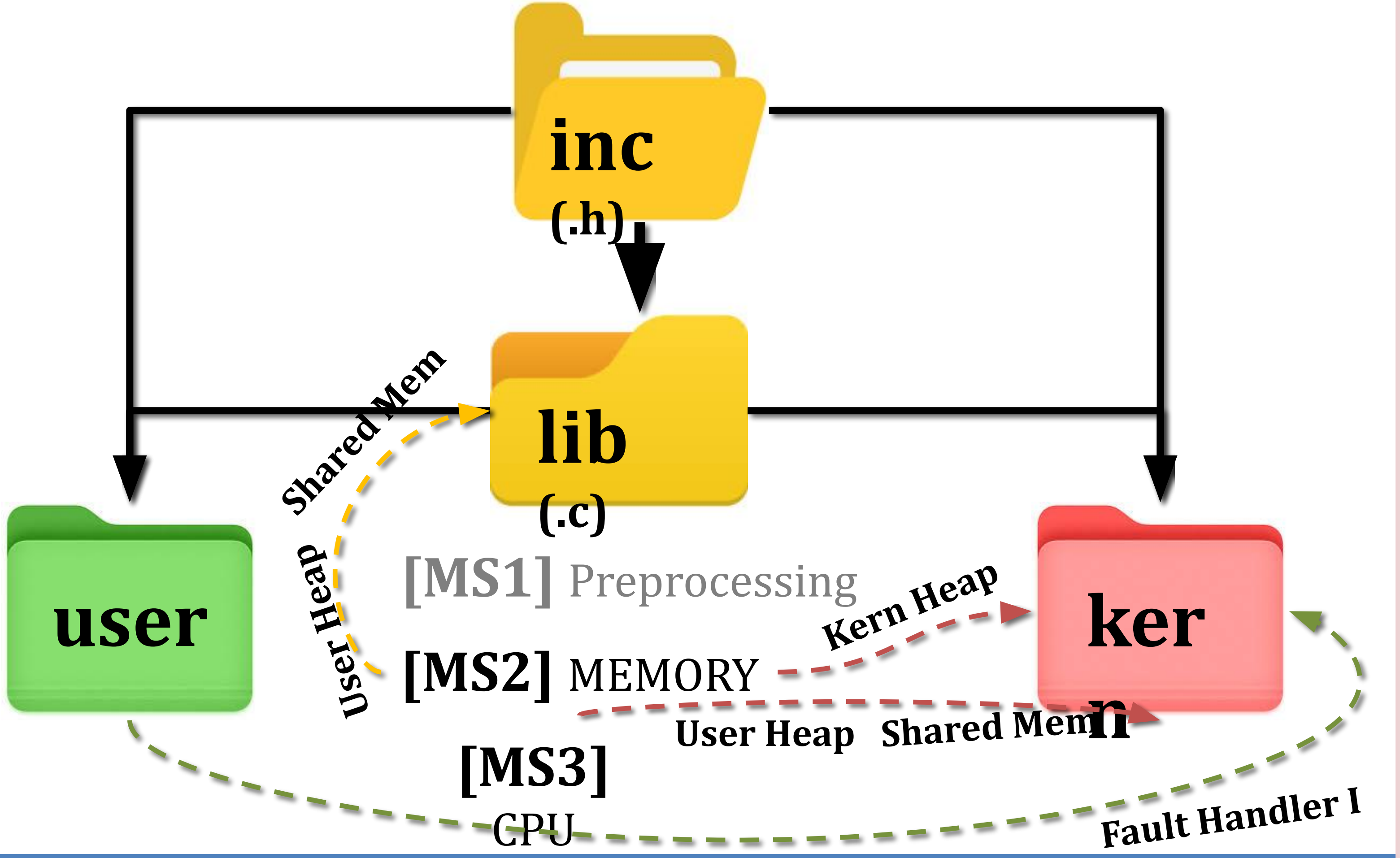
ADVICE#2: START immediately!

- To have the chance to ask and to understand errors in your code in whatever you want during your **mentor's support before the deadline**.

ADVICE#3: MUST read the ppt & doc CAREFULLY

- Detailed steps
- Helper ready made functions (*appendices*)

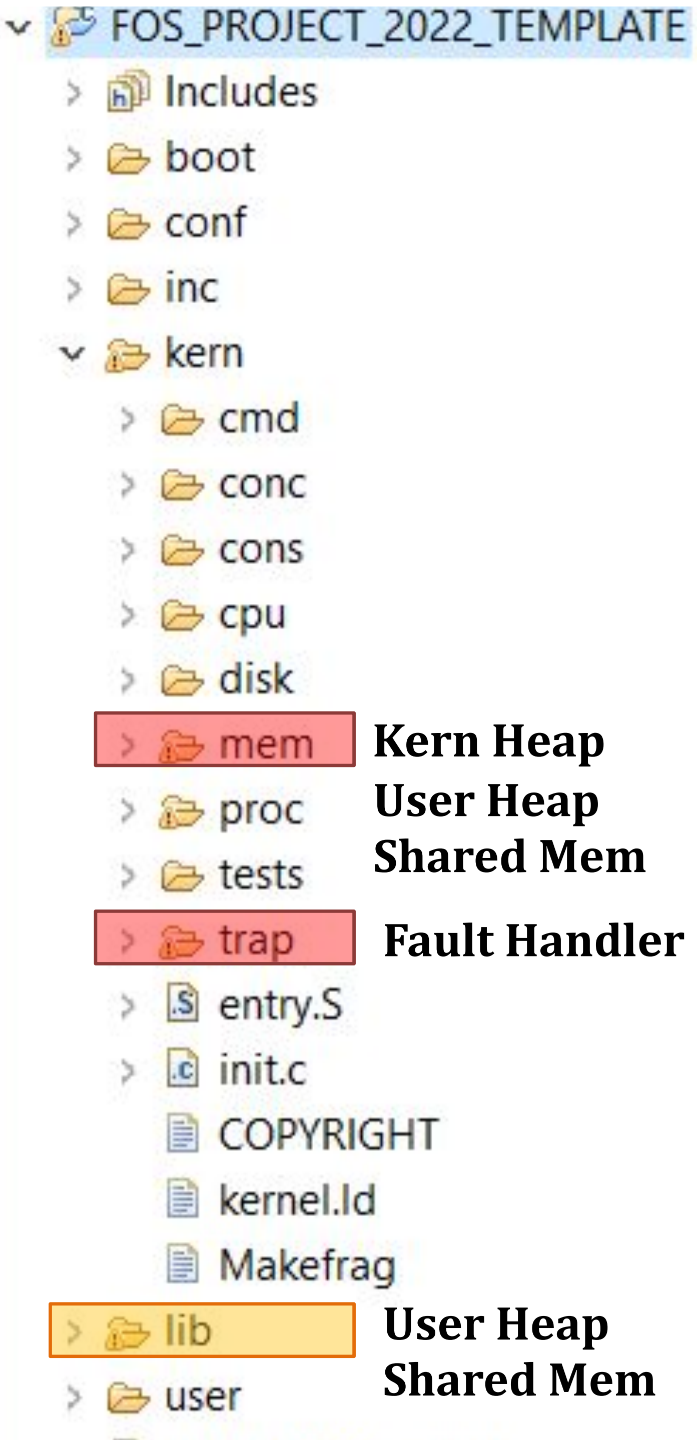
PROJECT BIG PICTURE



[MS1] Preprocessing

[MS2] MEMORY

[MS3] CPU



```
⌄ FOS_PROJECT_2022_TEMPLATE
  › Includes
  › boot
  › conf
  › inc
  ⌄ kern
    › cmd
    › conc
    › cons
    › cpu
    › disk
    › mem
    › proc
    › tests
    › trap
    › entry.S
    › init.c
      COPYRIGHT
      kernel.ld
      Makefrag
  › lib
  › user
```

mem: Kern Heap, User Heap, Shared Mem

trap: Fault Handler

lib: User Heap, Shared Mem

Code Updates

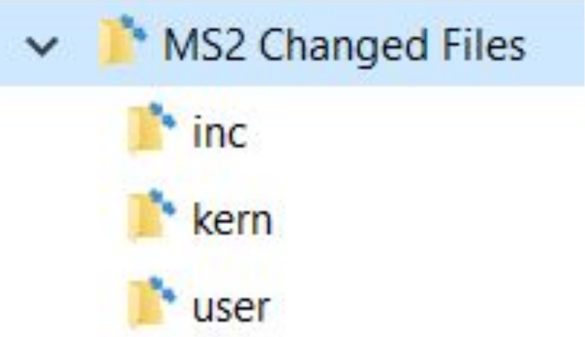
PART0: PREREQUISITES

New Files

1. **SELECT ALL** in the given “**Changed files**” folder,

2. **COPY** & **PASTE** (**REPLACE ALL**) in **FOS_CODES/FOS_PROJECT_2024_TEMPLATE/**

NOTE: If any of these files are already edited by you in MS1, make sure to apply the edits in the new files



- MS2 Changed Files
  - inc
  - kern
  - user

Given Codes

APPENDICES:

1. ENTRY MANIPULATION in TABLES and DIRECTORY
2. PAGE FILE HELPER FUNCTIONS
3. WORKING SET STRUCTURE & HELPER FUNCTIONS
4. MEMORY MANAGEMENT FUNCTIONS
5. COMMANDs

Given Codes

MEMORY MANAGEMENT FUNCTIONS: [Detailed Explanation in **Lab#3**]

Function	Description
`**PDX**(uint32 virtual address)`	Gets the page directory index in the given virtual address (10 bits from 22 – 31).
`**PTX**(uint32 virtual address)`	Gets the page table index in the given virtual address (10 bits from 12 – 21).
`**ROUNDUP**(uint32 value, uint32  align)`	Rounds a given "value" to the nearest upper value that is divisible by "align".
`**ROUNDDOWN** (uint32 value, uint32 align)`	Rounds a given "value" to the nearest lower value that is divisible by "align".
`**tlb_invalidate** (uint32* directory, uint32 virtual address)`	Refresh the cache memory (TLB) to remove the given virtual address from it.
`isKHeapPlacementStrategyFIRSTFIT()`	Check which strategy is currently selected using the given functions.

Given Codes

MEMORY MANAGEMENT FUNCTIONS: [Detailed Explanation in **Lab#4**]

Function Name	Description
allocate_frame	Used to allocate a free frame from the free frame list
free_frame	Used to free a frame by adding it to free frame list
map_frame	Used to map a single page with a given virtual address into a given allocated frame, simply by setting the directory and page table entries
get_page_table	Get a pointer to the page table if exist
create_page_table	Create a new page table by allocating a new page at the kernel heap, zeroing it and finally linking it with the directory
unmap_frame	Used to un-map a frame at the given virtual address, simply by clearing the page table entry
get_frame_info	Used to get both the page table and the frame of the given virtual address

CAUTION

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

REMEMBER: Ensure CORRECTNESS by DESIGN

Agenda

- Logistics
- Part 0: Code Updates
- **Part 1: Kernel Heap**
  - Block Allocator
  - Page Allocator
- Part 2: Fault Handler I
- Part 3: User Heap
  - Block Allocator
  - Page Allocator
- Part 4: Shared Memory
- Summary & Quick Guide
- How to submit?

Kernel Heap

The main functions required by MS2 to handle "**Kernel Heap**" are:

#	Function	File
1	**`Initialization`**	All essential declarations in: Kern/mem/kheap.h **Functions definitions <u>TO DO</u> in:** **Kern/mem/kheap.c**
2	**`sbrk()`**	
3	**`kmalloc (using FIRST FIT)`**	
4	**`kfree`**	
5	**`kheap_virtual_address`**	
6	**`kheap_physical_address`**	
MS2 BONUS 1	**`krealloc (using FIRST fit)`**	
MS2 BONUS 2	**`Fast Page Allocator`**	

Kernel Heap

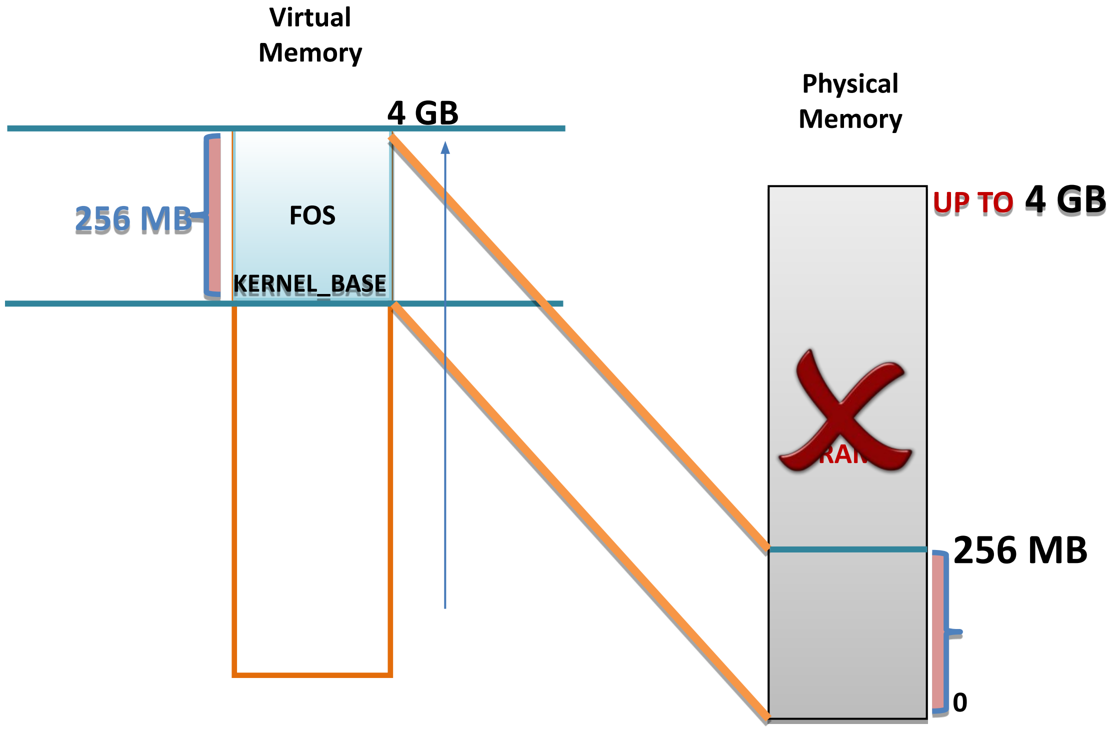
IMPORTANT NOTE

Before starting in the KHEAP functions, you MUST DO the following:

- Go to '**inc/memlayout.h**' and set **`USE_KHEAP`** by **1**

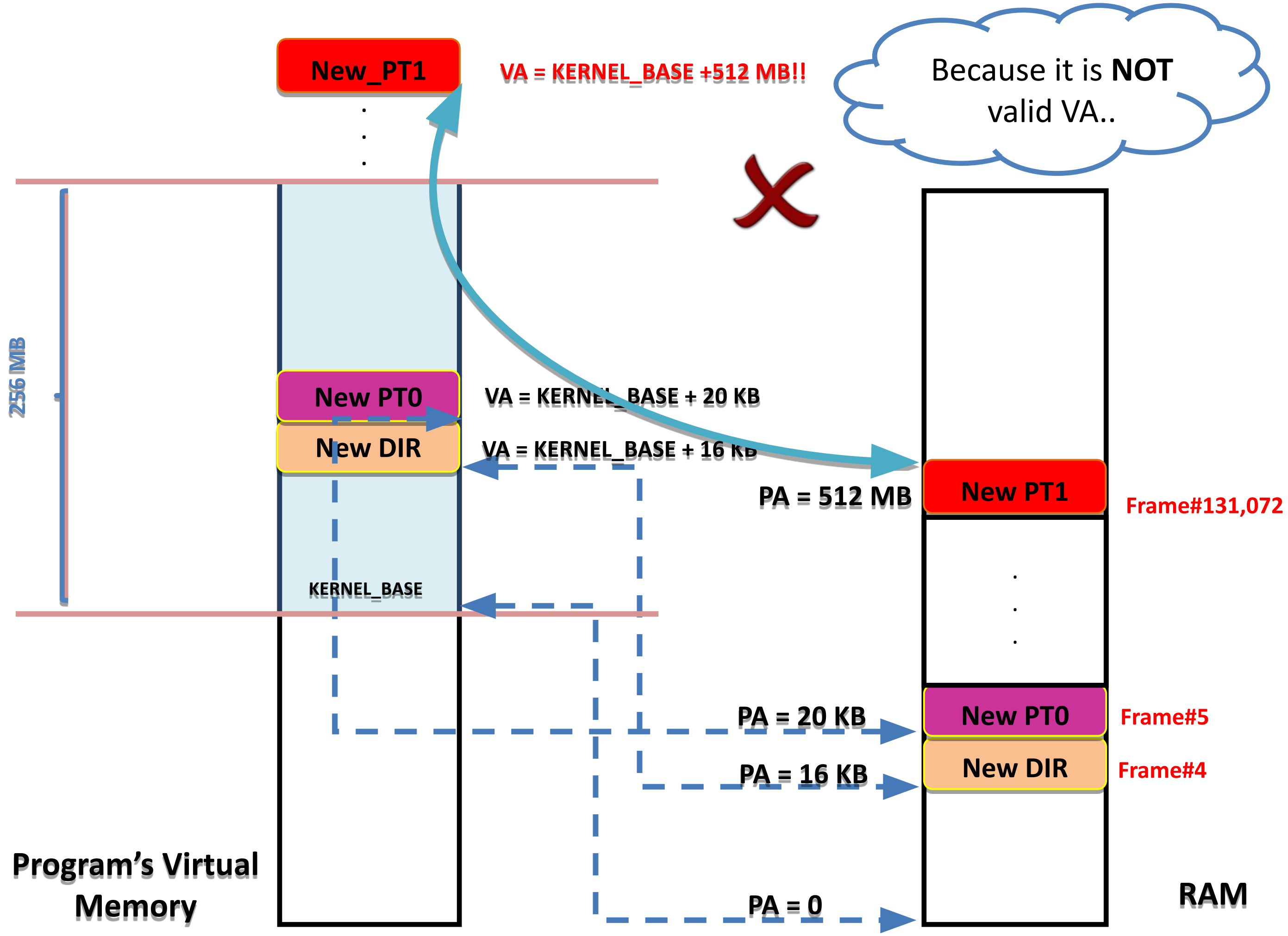
Kernel Heap – What is new?

Current:	Kernel is **one-to-one** mapped to 256 MB RAM
Problem:	Kernel can’t directly access beyond 256 MB RAM



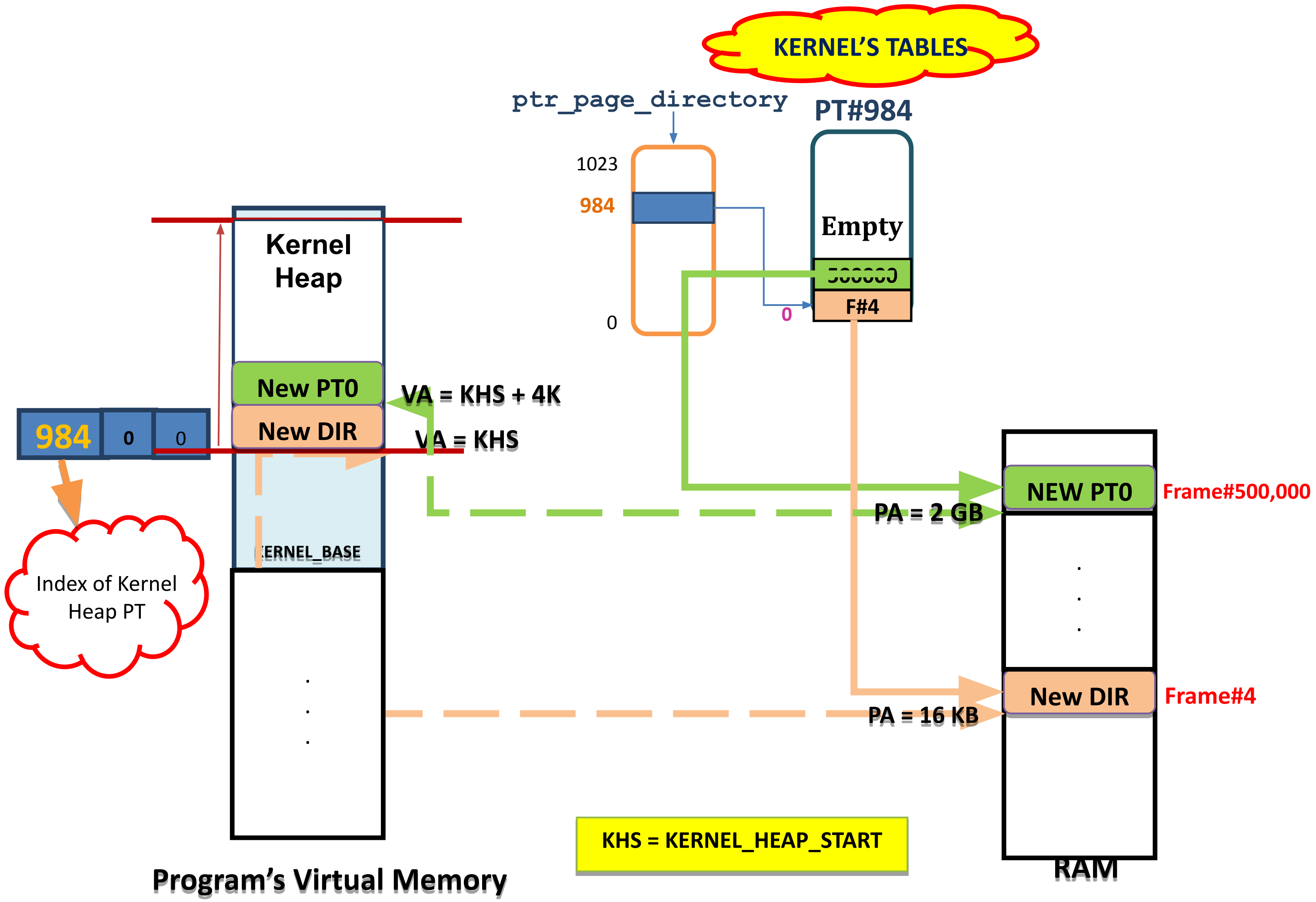
Kernel Heap – What is new?

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



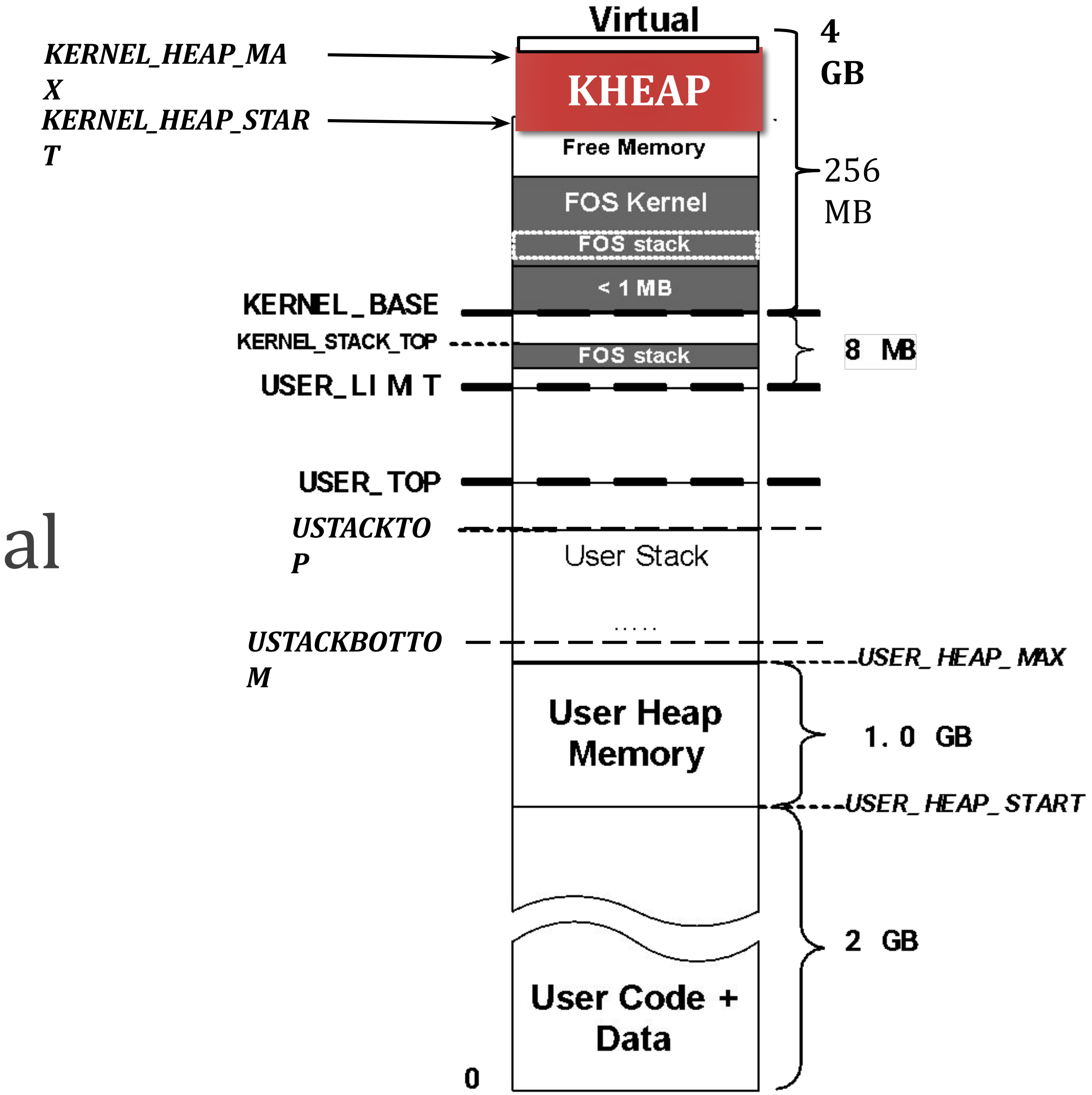
Kernel Heap – What is new?

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



Kernel Heap – What is new?

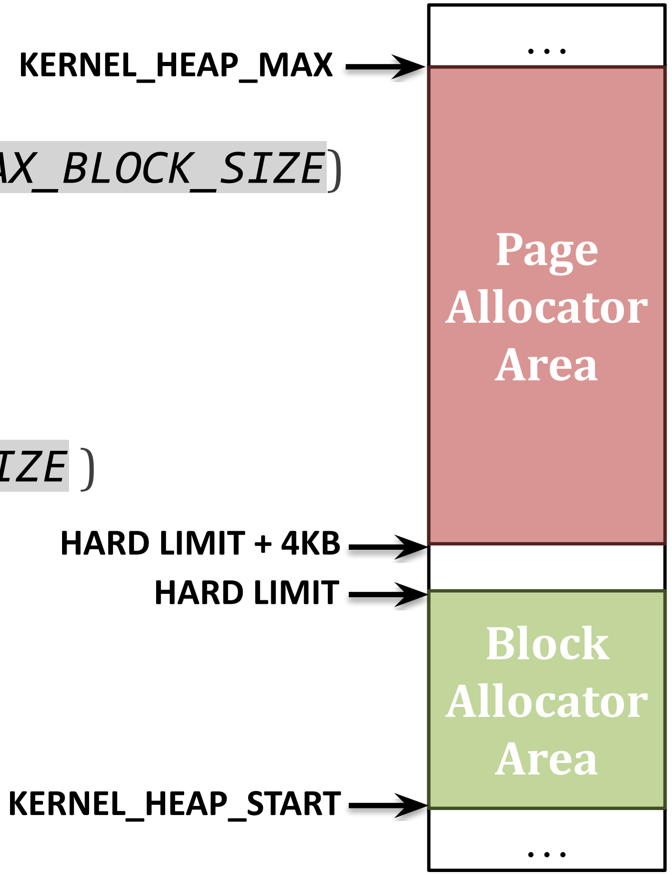
- Kernel Heap lies at the end of the virtual space



Kernel Heap – Allocation Types?

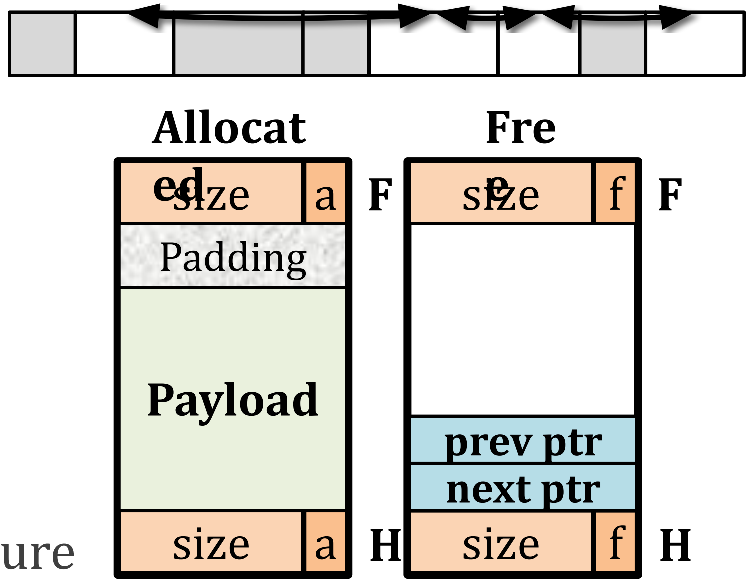
There're **TWO** types of allocator

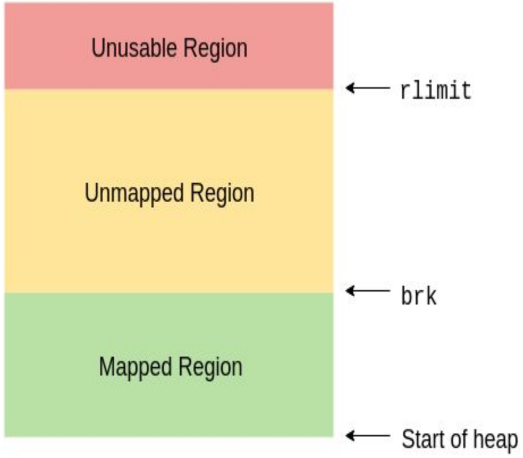
1. **Block Allocator**
    1. Used to allocate **small blocks** (with size **LESS OR EQUAL** *DYN_ALLOC_MAX_BLOCK_SIZE*)
    2. Use Dynamic Allocator from MS#1
    3. Range: **[**KERNEL_HEAP_START, HARD_LIMIT**)**
2. **Page Allocator**
    1. Used to allocate **chunk of pages** (with size > *DYN_ALLOC_MAX_BLOCK_SIZE*)
    2. Allocation is done on **page boundaries** (i.e. internal fragmentation)
    3. Range: **[**HARD_LIMIT + PAGE_SIZE, KERNEL_HEAP_MAX**)**

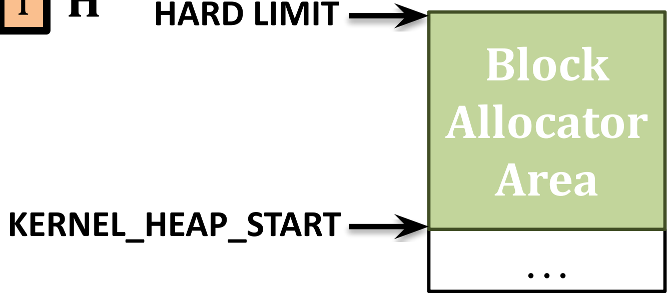


Kernel Heap – **Block Allocator**

1. Has 3 limits:
   1. **Start**: begin of the dynamic allocator area
   2. **Break**: end of current mapped area
   3. **Hard** Limit: which the break can't surpass
2. Break can only be changed using **`sbrk()`**
3. Use Dynamic Allocator with its data structure







#1: KH Block Alloc Initialization

Description:

- Need to **keep track** of 3 variables for the kernel dynamic allocator:
  1. **start**,
  2. segment **break** (end of the allocated space) and
  3. hard **limit** (max limit that can’t be exceeded).
- These should be declared in the **`kern/mem/kheap.h`**.
- **Initialize** the 3 variables, together with the dynamic allocator itself inside:

  `int` **`initialize_kheap_dynamic_allocator(...)`** defined in **`kern/mem/kheap.c`**
- This function, in turn, is **already called** inside the **`FOS_initialize`**() in **`init.c`**

#1: KH Block Alloc Initialization

```
int initialize_kheap_dynamic_allocator(uint32 daStart, uint32
                                      initSizeToAllocate, uint32 daLimit);
```

Description:

1. **Initialize** the block allocator of kernel heap with the given **start** address, **size** & **limit**
2. **All pages** in the given range should be **allocated** and **mapped**
3. **Remember**: call the **`initialize_dynamic_allocator(..)`** to complete the initialization

- **Return:**
  - On success: 0
  - Otherwise (if no memory OR initial size exceed the given limit): kernel should **`panic()`**

Testing:

- Will be tested during the other tests…

#2: sbrk()

```
void* sbrk(int numOfPages);
```

Description:

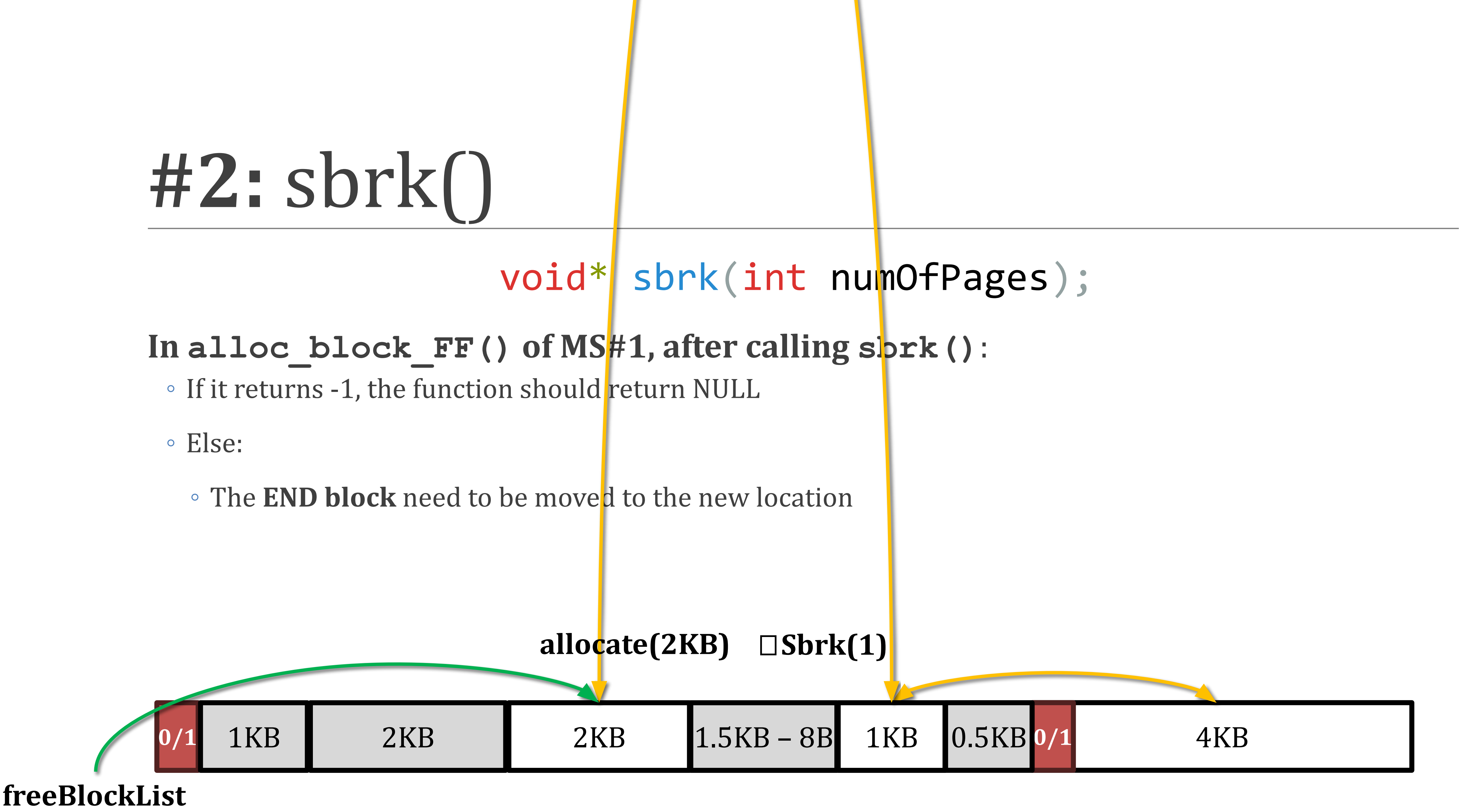
- Since virtual address space is mapped in quanta of **pages** (multiple of 4KB).
- **`sbrk`** always increase the size by **multiple of pages**
  1. If increment > 0: if within the **hard limit**
     1. **move** the segment break of the kernel to **increase** the size of its heap by the given `numOfPages`,
     2. **allocate** pages and map them into the kernel virtual address space as necessary,
     3. **returns** the address of the **previous break** (i.e. the beginning of newly mapped memory).
  2. If increment = 0: just return the current position of the segment break
  - if no memory OR break exceed the hard limit: it should **`return -1`**

#2: sbrk()

```
void* sbrk(int numOfPages);
```

In `alloc_block_FF()` of MS#1, after calling `sbrk()`:

- If it returns -1, the function should return NULL
- Else:
  - The **END block** need to be moved to the new location



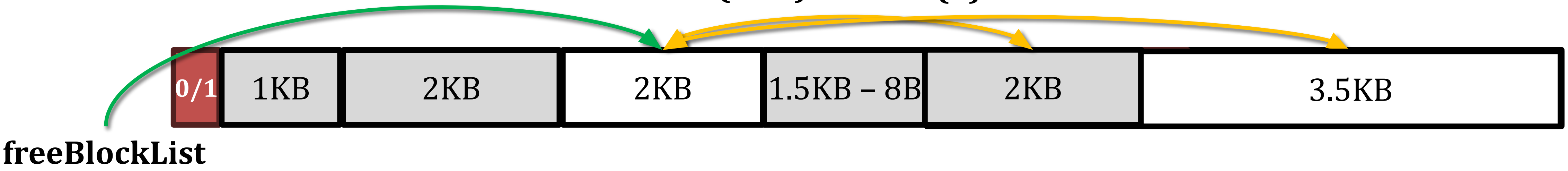
#2: sbrk()

```
void* sbrk(int numOfPages);
```

In `alloc_block_FF()` of MS#1, after calling `sbrk()`:

- If it returns -1, the function should return NULL
- Else:
  - The **END block** need to be moved to the new location
  - If there's a **free block** at the end of the old break, it should be **coalesced** with the new space
  - **Allocate** the required space

allocate(2KB) □Sbrk(1)



#2: sbrk()

```
void* sbrk(int numOfPages);
```

In `alloc_block_FF()` of MS#1, after calling `sbrk()`:

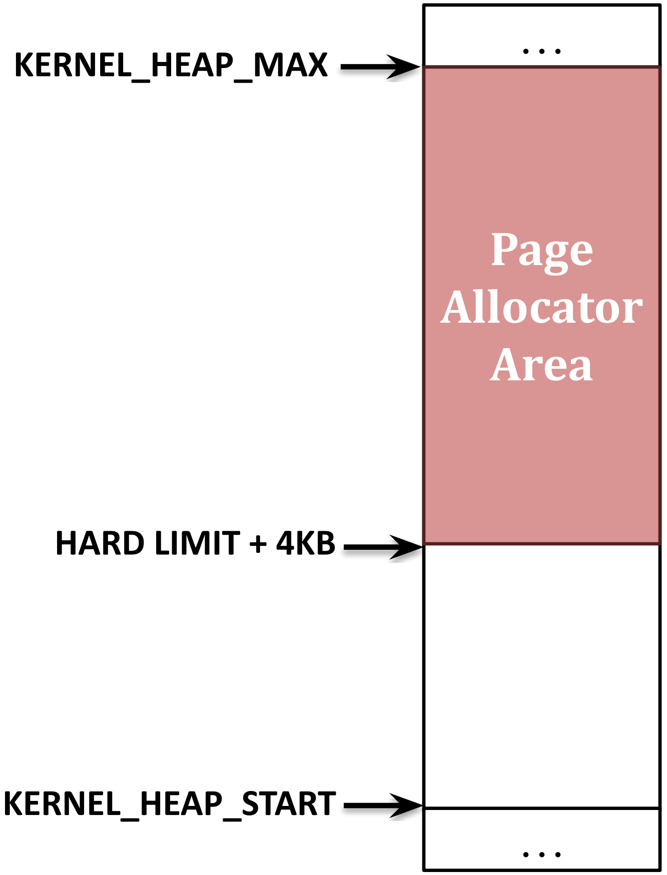
- If it returns -1, the function should return NULL
- Else:
  - The **END block** need to be moved to the new location
  - If there's a **free block** at the end of the old break, it should be **coalesced** with the new space
  - **Allocate** the required space

Testing:

`FOS> tst kheap FF sbrk` □ tests sbrk & the changes in `alloc_block_FF`

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: **FIRST FIT**



Kernel Heap – **Page Allocator**

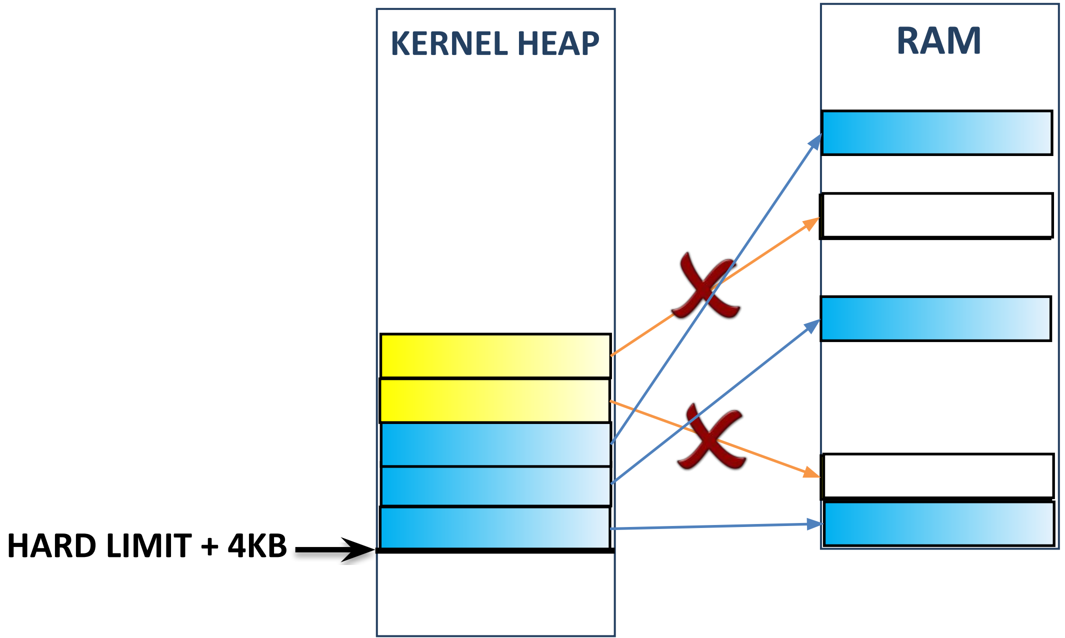
1. **Kmalloc():** dynamically allocate space
2. **Kfree():** delete a previously allocated space

kmalloc()

ALL Pages Created BUT NOT Tables

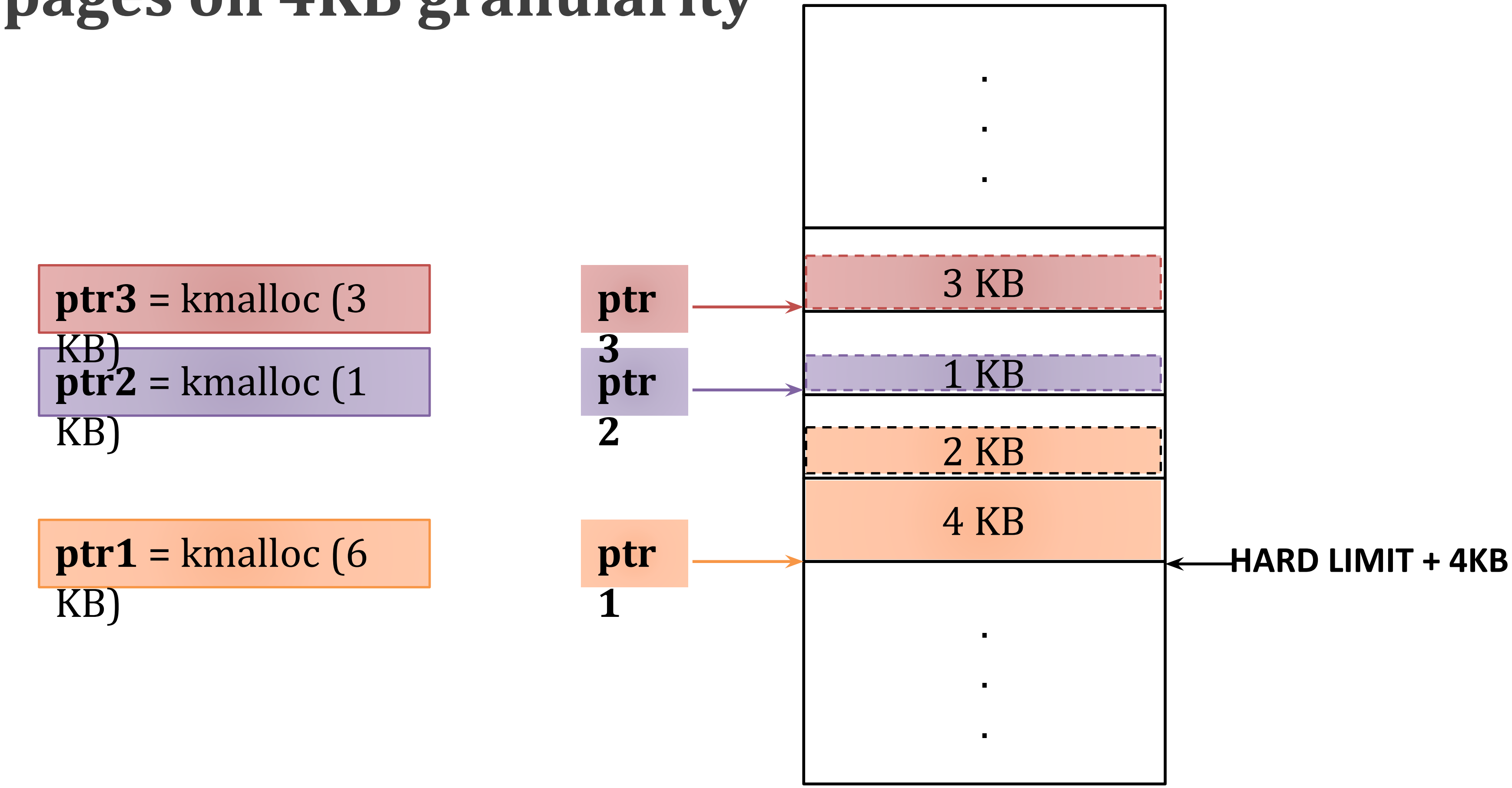
Kfree()

Remove All Pages From Memory



Kernel Heap – **Page Allocator**

Allocate pages on 4KB granularity



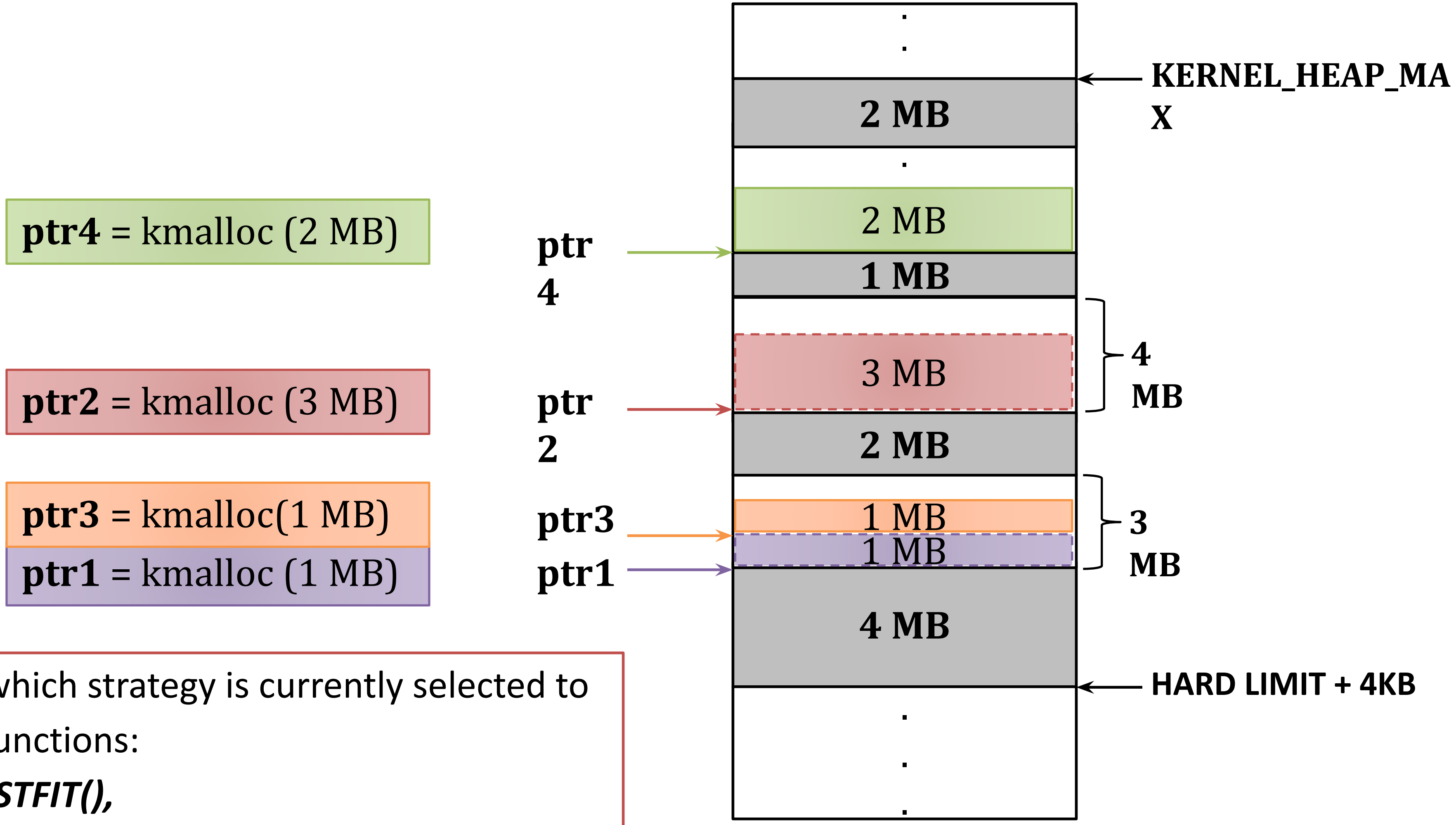
ptr3 = kmalloc (3 KB)

ptr2 = kmalloc (1 KB)

ptr1 = kmalloc (6 KB)

Kernel Heap – Page Allocator

FIRST FIT Strategy



In kmalloc, you need to check which strategy is currently selected to apply its code using the given functions:

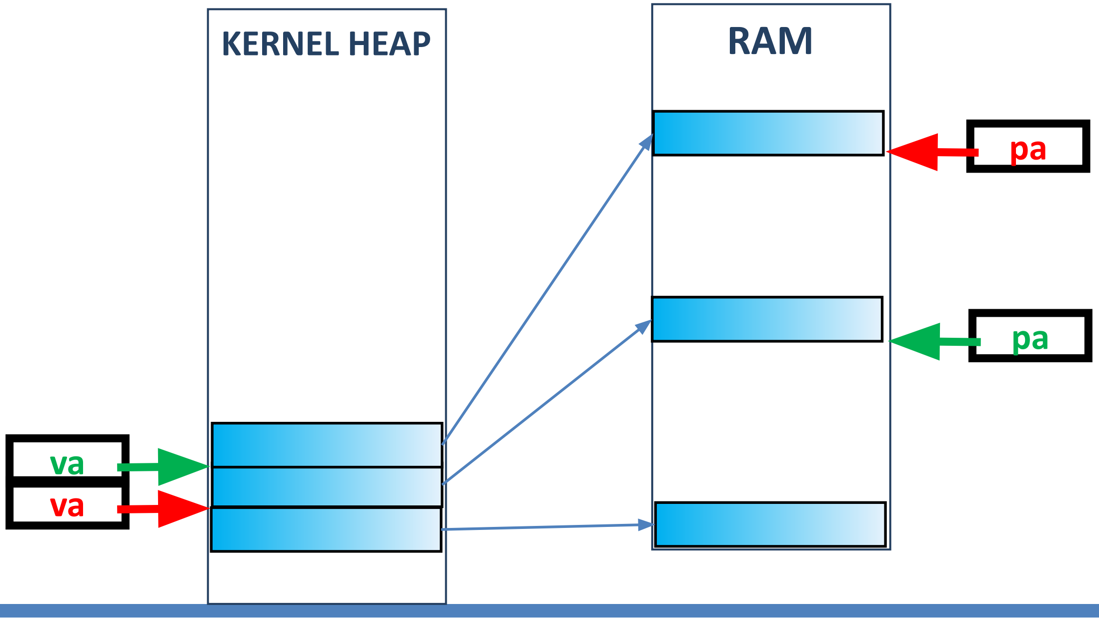
isKHeapPlacementStrategyFIRSTFIT(),

isKHeapPlacementStrategyBESTFIT(), …

Kernel Heap – **Page Allocator**

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

IMP: Both MUST be Efficient ~O(1)



#3: kmalloc()

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

Description:

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

Testing:

1. **`FOS> tst kheap FF kmalloc 1`** □ **tests allocation only**
2. **`FOS> tst kheap FF kmalloc 2`** □ **tests FF strategy#1 [PAGE Alloc.] (depends on kfree)** [always FIT]
3. **`FOS> tst kheap FF kmalloc 3`** □ **tests FF strategy#2 [PAGE & BLOCK] (depends on kfree)** [FIT & NOT]

#4: 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
4. Else (i.e. invalid address): should **`panic(…)`**

Testing:

```
FOS> tst kheap FF kfree
```

#5: 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)**

Testing:

1. **`FOS> tst kheap FF kphysaddr`**

#6: 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)**

Testing:

```
FOS> tst kheap FF kvirtaddr
```

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 for both **[BLOCK ALLOCATOR]** and **[PAGE ALLOCATOR]**

Testing:

- **[UNSEEN] test at your own**

BONUS#2: Fast Page Allocator

Description:

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

Testing:

```
FOS> tst kheap FF fast
```

[Should run in LESS THAN 5 sec]

"test [TEST NAME] completed. Evaluation = …%”
To ensure the success of a test, this message like this **MUST be appeared without any ERROR messages or PANICs**.

Kernel Heap – **Testing**

- □ Test each function independently in a **FRESH SEPARATE RUN**.

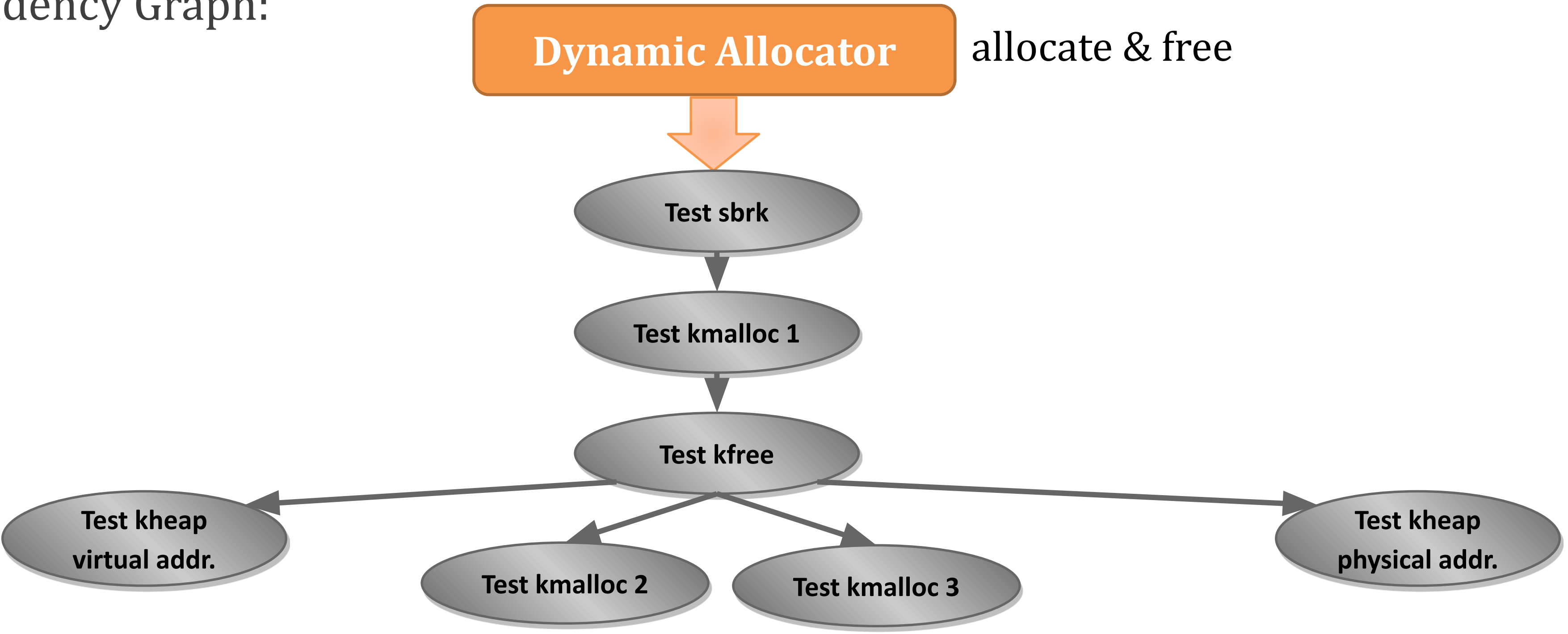
- □ The time limit of each individual test: **max of 15 sec / each**

- □ Before testing any of the kheap functions: `Go to 'inc/memlayout.h' and set USE_KHEAP by 1`

Function	Testing	Files
Initialization	Will be tested during the other tests	kern/mem/kheap.h & .c
sbrk()	**FOS>** tst kheap FF sbrk □ tests sbrk & allocate	kern/mem/kheap.c
kmalloc (FIRST FIT)	1.**FOS>** tst kheap FF kmalloc 1 □ tests allocation only 2.**FOS>** tst kheap FF kmalloc 2 □ tests FF in PAGE Alloc 3.**FOS>** tst kheap FF kmalloc 3 □ tests FF in PAGE & BLK 2 & 3 depend on kfree	
kfree	**FOS>** FOS> tst kheap FF kfree	
kheap_virtual_address	**FOS>** FOS> tst kheap FF kvirtaddr	
kheap_physical_address	**FOS>** FOS> tst kheap FF kphysaddr	

Kernel Heap – **Testing**

- Dependency Graph:

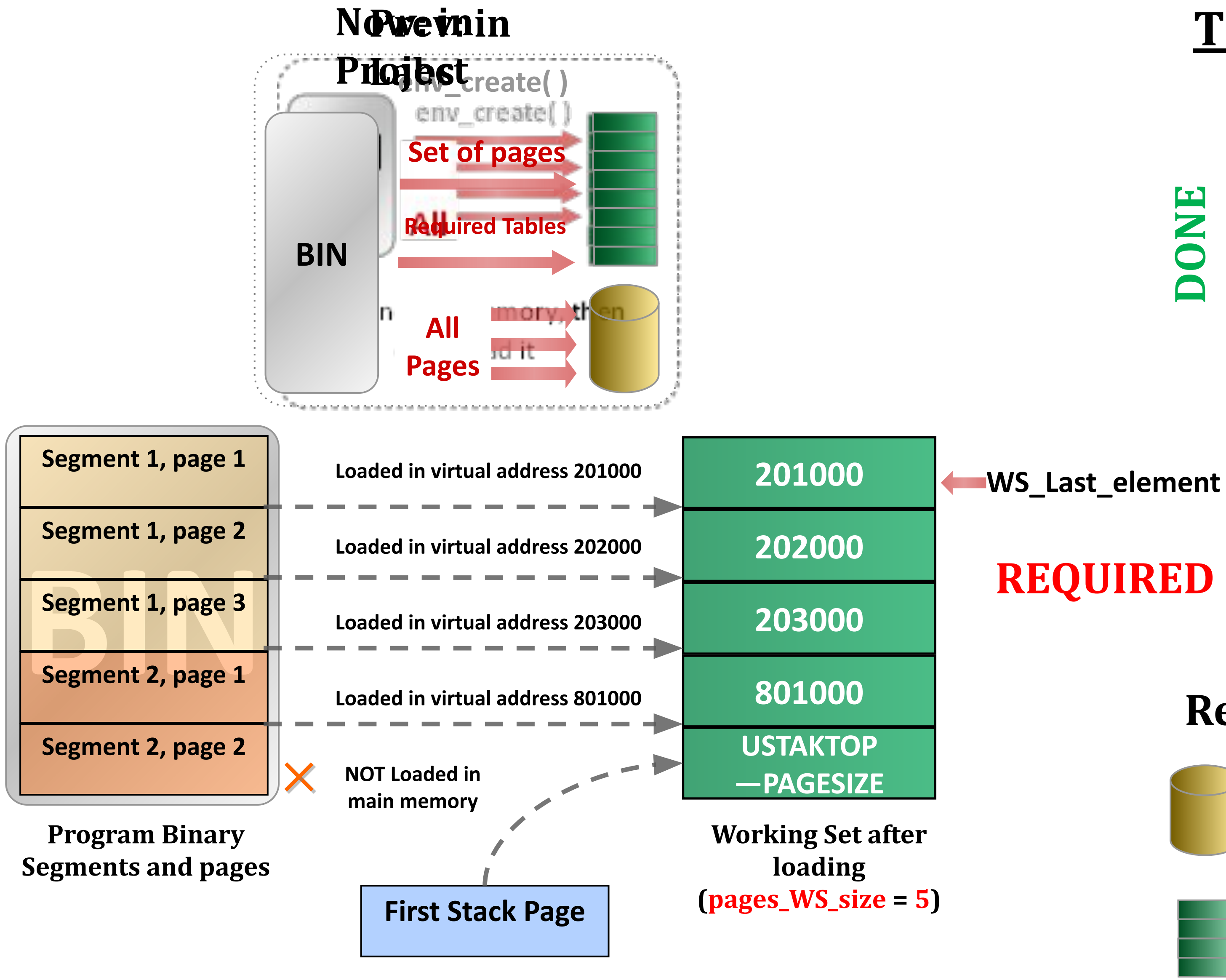


REMEMBER: This module MUST be FINISHED 1st

Agenda

- Logistics
- Part 0: Code Updates
- **Part 1: Kernel Heap**
  - Block Allocator
  - Page Allocator
- **Part 2: Fault Handler I**
- Part 3: User Heap
  - Block Allocator
  - Page Allocator
- Part 4: Shared Memory
- Summary & Quick Guide
- How to submit?

Load Program [env_create]


Now in Project
env_create( )
BIN
Set of pages
Required Tables
All Pages
Segment 1, page 1
Segment 1, page 2
Segment 1, page 3
Segment 2, page 1
Segment 2, page 2
Loaded in virtual address 201000
Loaded in virtual address 202000
Loaded in virtual address 203000
Loaded in virtual address 801000
NOT Loaded in main memory
201000
202000
203000
801000
USTAKTOP —PAGESIZE
WS_Last_element
First Stack Page
Program Binary Segments and pages
Working Set after loading (pages_WS_size = 5)

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.

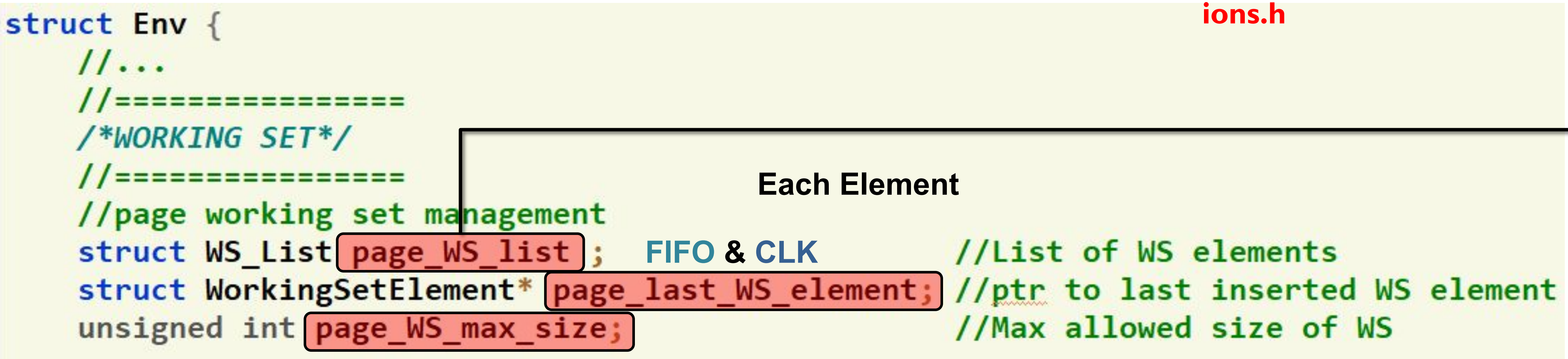
REQUIRED

Refer to APPENDICES for:

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

Working Set: **Structure**

inc/environment_definitions.h



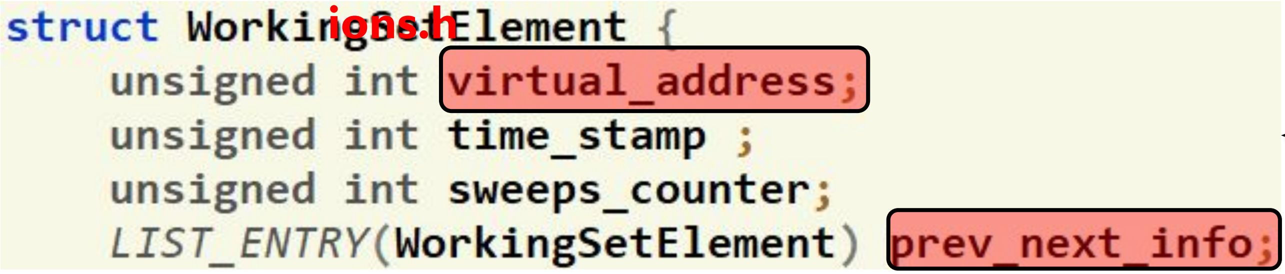
```
struct Env {
    //...
    //================
    /*WORKING SET*/
    //================
    //page working set management
    struct WS_List page_WS_list ;                    //List of WS elements
    struct WorkingSetElement* page_last_WS_element; //ptr to last inserted WS element
    unsigned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

Each Element

FIFO & CLK

Proc Limit

inc/environment_definitions.h



```
struct WorkingSetElement {
    unsigned int virtual_address;
    unsigned int time_stamp ;
    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**]

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

- 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)
```

- 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()`)

#7: Kernel Dynamic Allocations 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()`**

Testing:

- **Will be tested during the next tests…**

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

Description:

1. **Create** a new object of **`struct WorkingSetElement`**
2. **Initialize** it by the given virtual address

Return

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

Testing:

- **Already tested in next placement test**

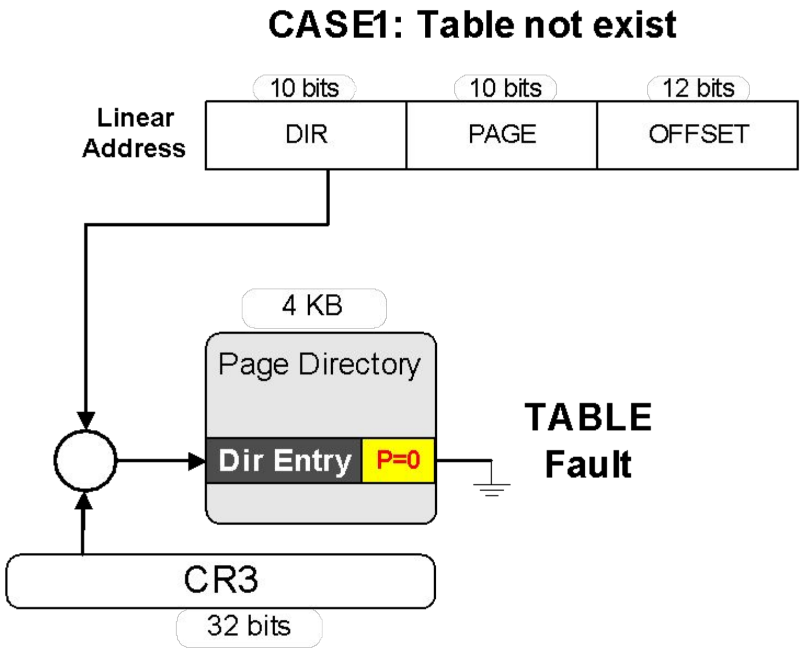
Fault Handler I: **Overview**

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** is 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



#8: Check Invalid Pointers

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

Description:

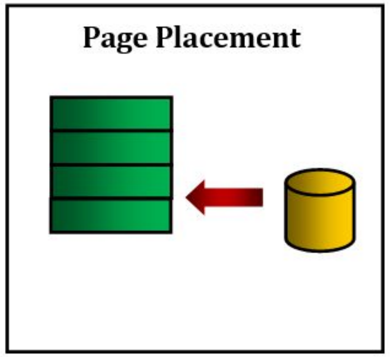
- **Validate** the **`faulted_va`** to ensure that it is :

1. **NOT** pointing to **UNMARKED** page in user heap
2. **NOT** pointing to **kernel**
3. **Exist** but with **read-only** permissions

- If **invalid**: it must be rejected without harm to the kernel or other running processes, by **exiting** the process using **env_exit()**

Testing:

```
FOS> run tia 15
```



#9: Placement

```
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)

Testing:

```
FOS> run tpp 20
```

NOTE: Check MS2 **appendices** to handle either the **working set** or the **page file** using some ready-made functions.

Fault Handler I: **Testing**

- Test each function in MS3 independently in a **FRESH SEPARATE RUN**

- The time limit of each individual test: **max of 10 sec / each**

```
"test [TEST NAME] completed. Evaluation = …%”
```

To ensure the success of a test, this message like this **MUST be appeared without any ERROR messages or PANICs**.

#	Test Functionality	Test
1	***tst_placement.c (tpp):*** tests page faults on stack + page placement	□ **FOS>** run tpp 20
2	***tst_invalid_access.c (tia):*** tests invalid pointers	□ **FOS>** run tia 15

Agenda

- Logistics
- Part 0: Code Updates
- **Part 1: Kernel Heap**
  - Block Allocator
  - Page Allocator
- **Part 2: Fault Handler I**
- **Part 3: User Heap**
  - Block Allocator
  - Page Allocator
- Part 4: Shared Memory
- Summary & Quick Guide
- How to submit?

User Heap

The main functions required by MS2 to handle “**User Heap**” are:

#	Function	File
1	**`Initialization`**	All essential declarations in: inc/uheap.h **Functions definitions TO DO in: lib/uheap.c**
2	**`sys_sbrk()`**	
3	**`malloc() (using FIRST FIT) [USER SIDE]`**	
4	**`free() [USER SIDE]`**	
5	**`allocate_user_mem [KERNEL SIDE]`**	**Kern/mem/chunk_operations.c**
6	**`free_user_mem [KERNEL SIDE]`**	
MS2 BONUS 3	**`free_user_mem: O(1) for removing pages from WS List`**	

User Heap

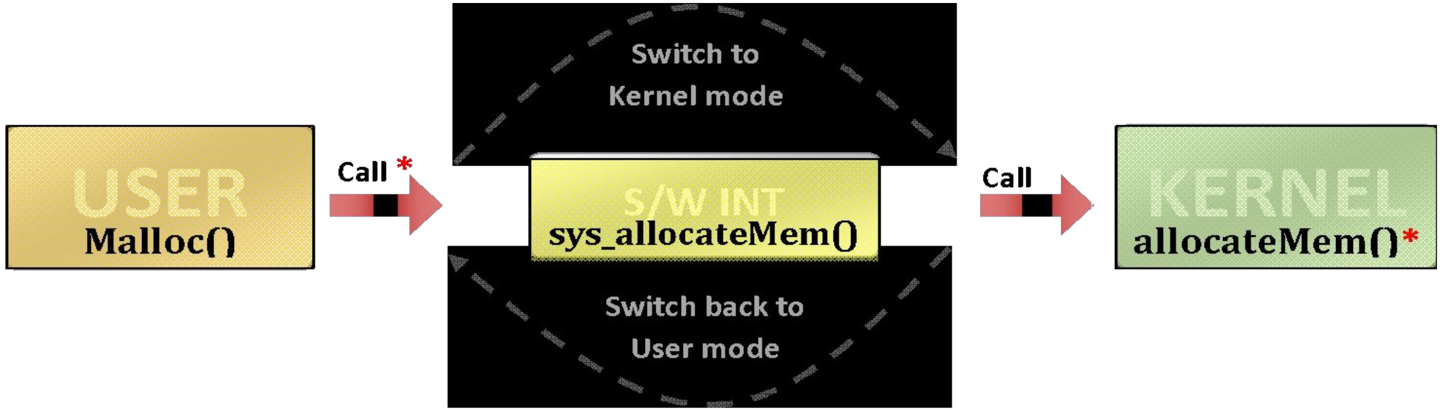
IMPORTANT NOTE

ALL Functions depend on the Implementation of Page Fault

PLACEMENT

User Heap: **Overview**

- **Before we start!**
  - Program runs in user mode (less privileges)
  - It requires functions from the kernel
  - So, need to switch to kernel mode, call the function, then return to user mode
  - SYSTEM CALLS (S/W interrupts) do this job!



NOTE: You should do the (*) operations only

User Heap: **Overview**

- **WHY?**
  - Program need **dynamic** allocations for its normal work
  - **De-allocations** are necessary after finishing using allocated memory:
  - virtual address space **external fragmentation** happens
  - **Minimize** these **fragmentations** as possible

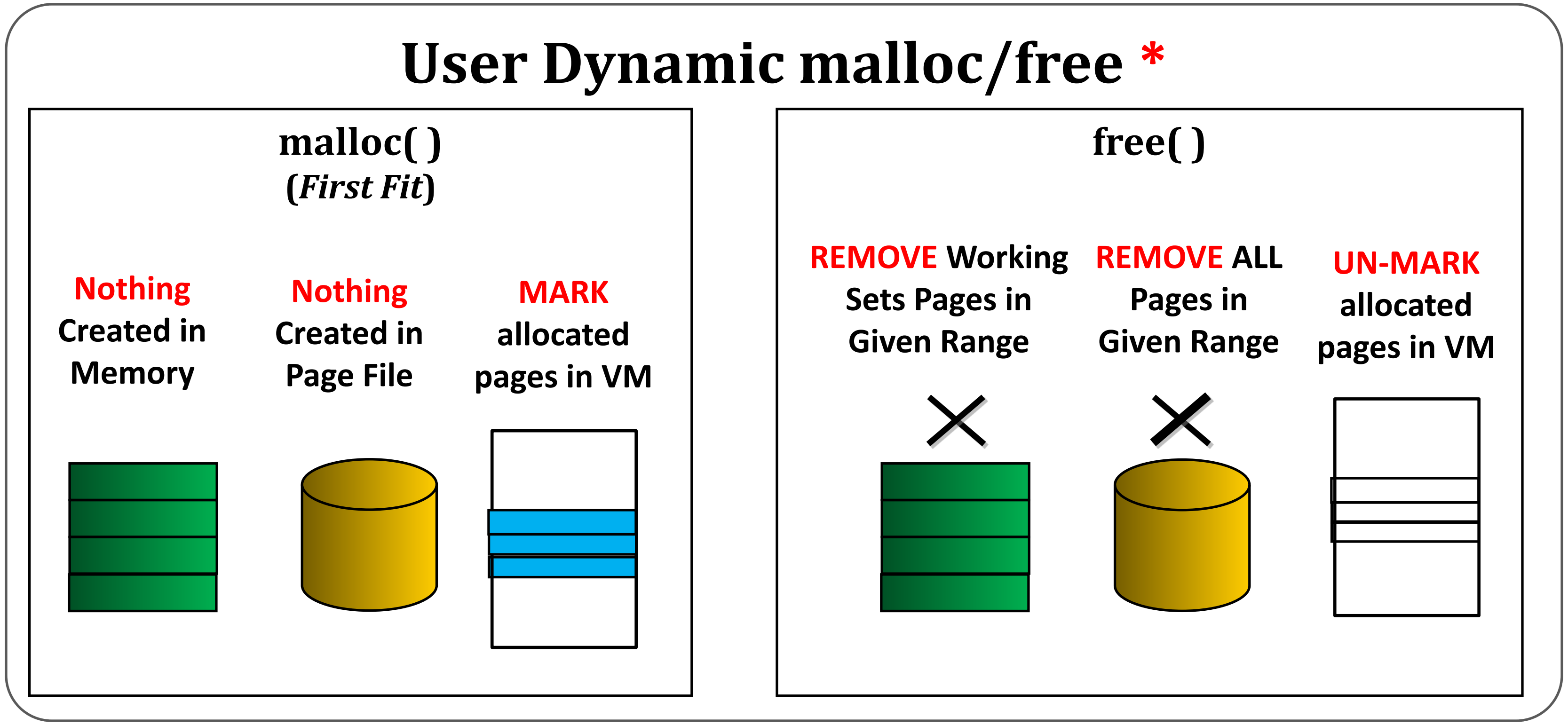
User Heap: **Overview**

- **Allocation**
  - **Example 1 (C++ and C):**
    - C++:   int * ptr_value = **new** int;
    - C: int * ptr_value = **malloc**(sizeof(int));
    - allocate 1 int (4 bytes) in virtual memory and return the allocated virtual address to “ptr_value”
  - **Example 2 (C++ and C):**
    - C++: float* arr = **new** float[200];
    - C: float* arr = **malloc**(sizeof(float) * 200);
    - allocate 200 floats (800 bytes) in memory and return the allocated address to “arr”

User Heap: **Overview**

- **De-allocation**
  - **Example 1 (C++ and C):**
    - C++: **delete** ptr_value;
    - C: **free**(ptr_value);
    - deallocate (free) 1 int (4 bytes) from virtual memory at address "ptr_value"
  - **Example 2 (C++ and C):**
    - C++: **delete**[] arr;
    - C: **free**(arr);
    - de-allocate (free) 200 floats (800 bytes) from virtual memory at address "arr"

User Heap: **Overview**



User Dynamic malloc/free *

malloc( )
(*First Fit*)

Nothing Created in Memory

Nothing Created in Page File

MARK allocated pages in VM

free( )

REMOVE Working Sets Pages in Given Range

REMOVE ALL Pages in Given Range

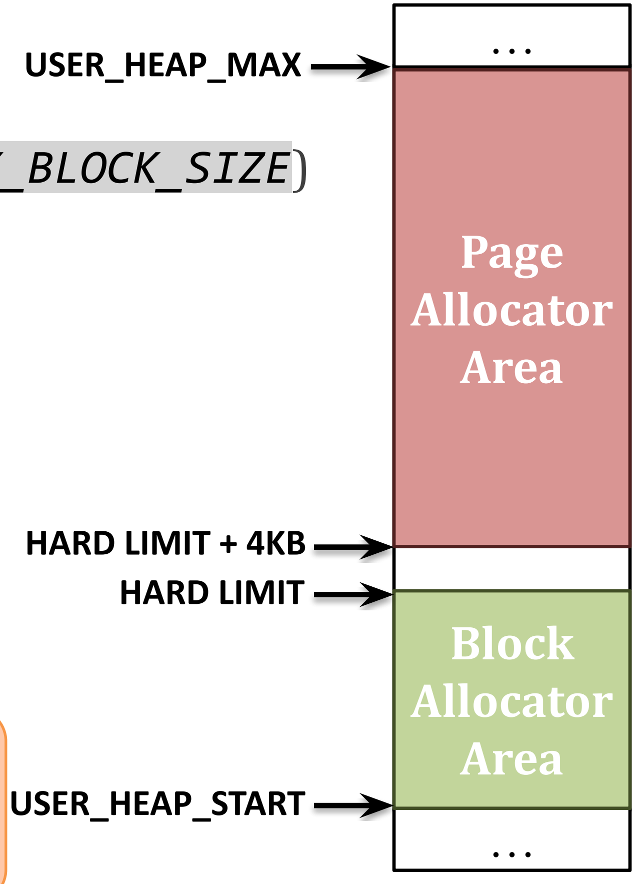
UN-MARK allocated pages in VM

User Heap – Allocation **Types**?

There're **TWO** types of allocator

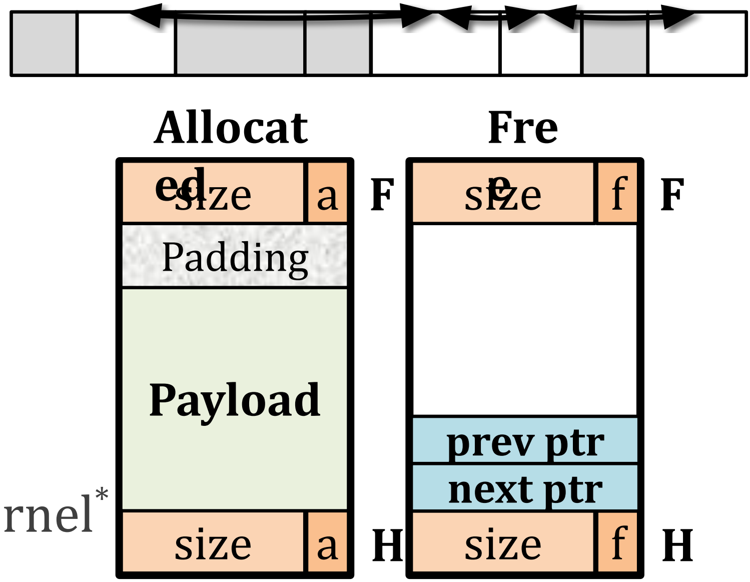
1. **Block Allocator**
   1. Used to allocate **small blocks** (with size **LESS OR EQUAL** *DYN_ALLOC_MAX_BLOCK_SIZE*)
   2. Use Dynamic Allocator from MS#1
   3. Use `sys_sbrk()` to extend the mapped area
   4. Range: [`USER_HEAP_START`, HARD_LIMIT)
2. **Page Allocator**
   1. Used to allocate **chunk of pages** (size > *DYN_ALLOC_MAX_BLOCK_SIZE*)
   2. Allocation is done on **page boundaries** (i.e. internal fragmentation)
   3. Range: [HARD_LIMIT + `PAGE_SIZE`, `USER_HEAP_MAX`)

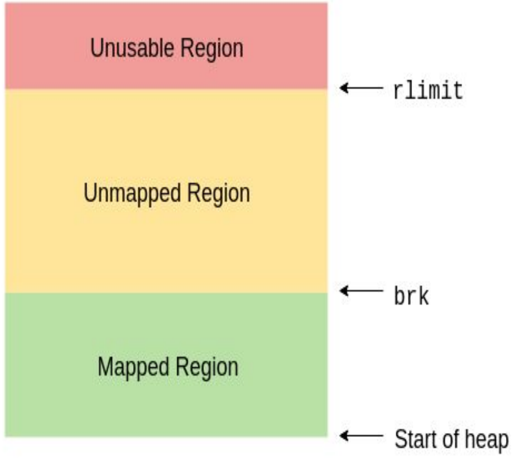
REMEMBER: In both, **NOTHING** is actually **allocated in RAM until** the **user access** it. In this case, allocation will be done via **Fault Handler**

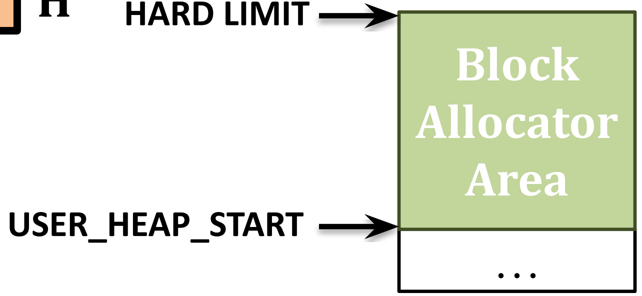


User Heap – **Block Allocator**

1. Has 3 limits:
   1. **Start**: begin of the dynamic allocator area
   2. **Break**: end of current mapped area
   3. **Hard** Limit: which the break can't surpass
2. Break can only be changed using **`sbrk()`** which already calls **`sys_sbrk()`** from Kernel*
3. Use Dynamic Allocator with its data structure







*:check the **`sbrk()`** in `lib/uheap.c`

#10: UH Block Alloc Initialization

Description:

- Need to keep track of 3 variables for the user block allocator:
  1. **start**,
  2. segment **break** (end of the allocated space) and
  3. hard **limit** (max limit that can't be exceeded).
- These should be declared in the `struct Env` defined in `inc/environment_definitions.h`.
- Initialize the 3 variables, together with the dynamic allocator itself inside:

  `initialize_uheap_dynamic_allocator(...)` defined in `kern/proc/user_environment.c`
- This function, in turn, is already called inside the `initialize_environment()` in `init.c`.
- **REMEMBER:** the **initial size** of the user block allocator should be **0**.

#10: UH Block Alloc Initialization

```
void initialize_uheap_dynamic_allocator(struct Env* e, uint32 daStart, uint32 daLimit)
```

Description:

1. **Initialize** the block allocator of user heap of the given **environ. “e”** with the given **start** & **limit**
2. **Call** the **`initialize_dynamic_allocator(..)`** to complete the initialization

- **REMEMBER:** there's **no initial allocations** for the block allocator of the user heap.

Testing:

- Will be tested during the other tests…

#11: sys_sbrk()

```
void* sys_sbrk(int numOfPages);
```

Description:

- Since virtual address space is mapped in quanta of **pages** (multiple of 4KB).
- **`sbrk`** always increase the size by **multiple of pages**
  1. If increment > 0: if within the **hard limit**
     1. **move** the segment break of the current user environment to **increase** the size of its block allocator,
     2. **allocate NOTHING**,
     3. **returns** the address of the **previous break** (i.e. the beginning of newly mapped memory).
  2. If increment = 0: just return the current position of the segment break
  - if no memory OR break exceed the hard limit: it should **`return -1`**

#11: sys_sbrk()

```
void* sys_sbrk(int numOfPages);
```

Notes:

- As in real OS, allocate pages **lazily**. While sys_sbrk **moves** the segment **break**, pages are **not allocated** until the user program tries to access data (i.e. will be allocated via **fault handler**).
- If **failed** to allocate additional pages for a user block allocator, for example,
  - the free frames are exhausted, or
  - the break exceed the limit of the block allocator.

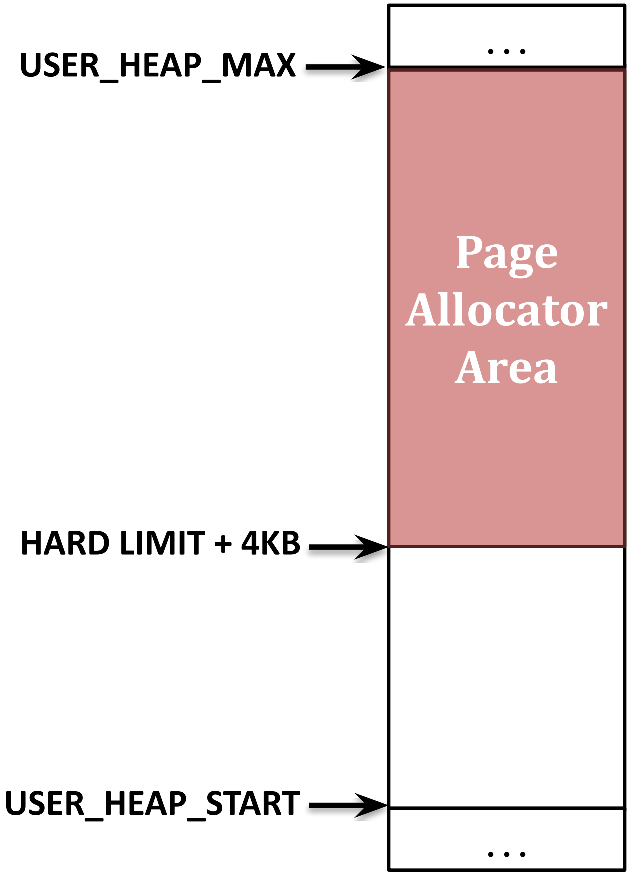
  function should `return -1`

Testing:

- **[UNSEEN] test at your own**

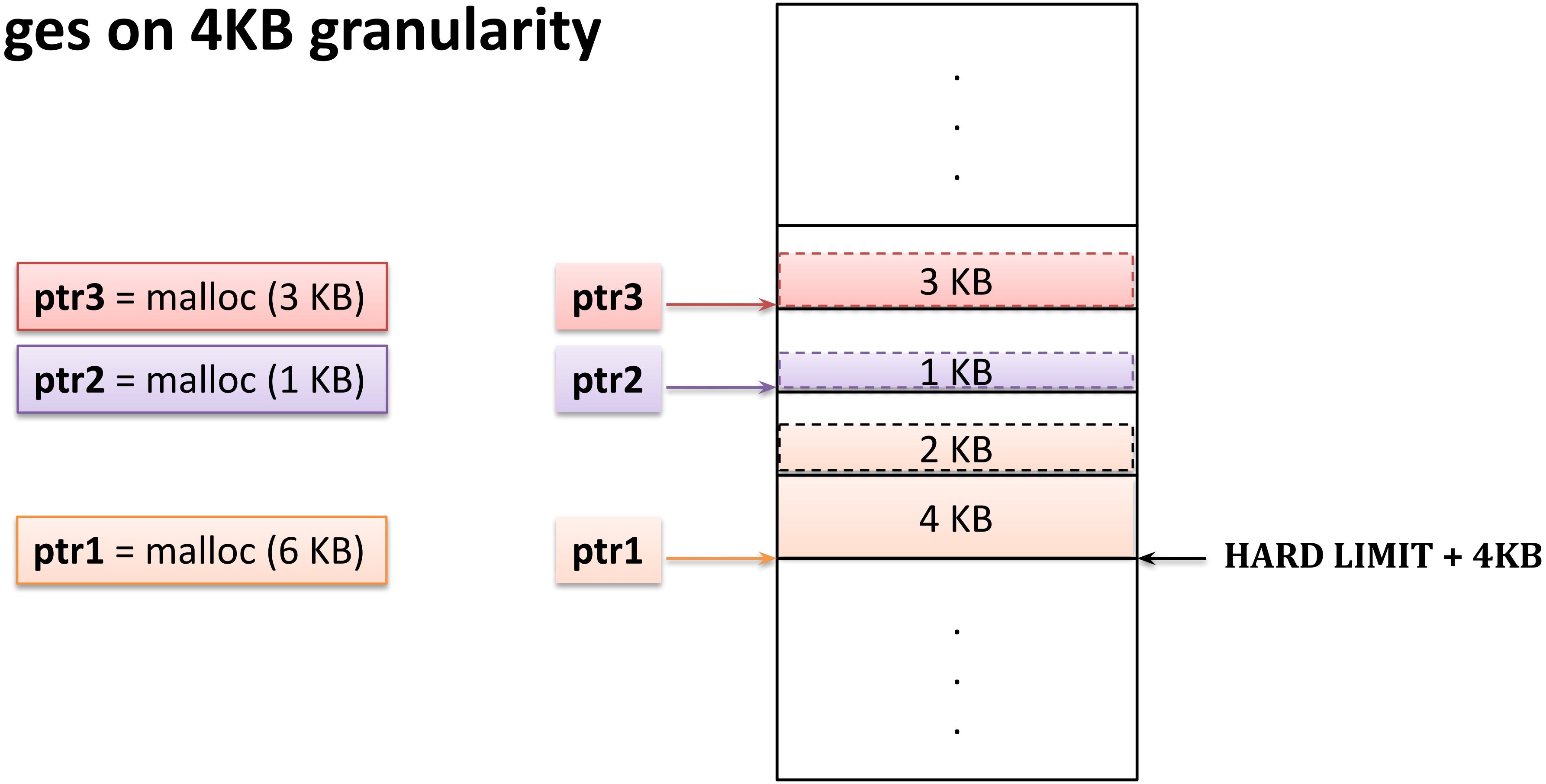
User Heap – **Page Allocator**

- 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
- **NO pages** will be **allocated** in **RAM** or **Page File**
- Allocation Strategy: **FIRST FIT**



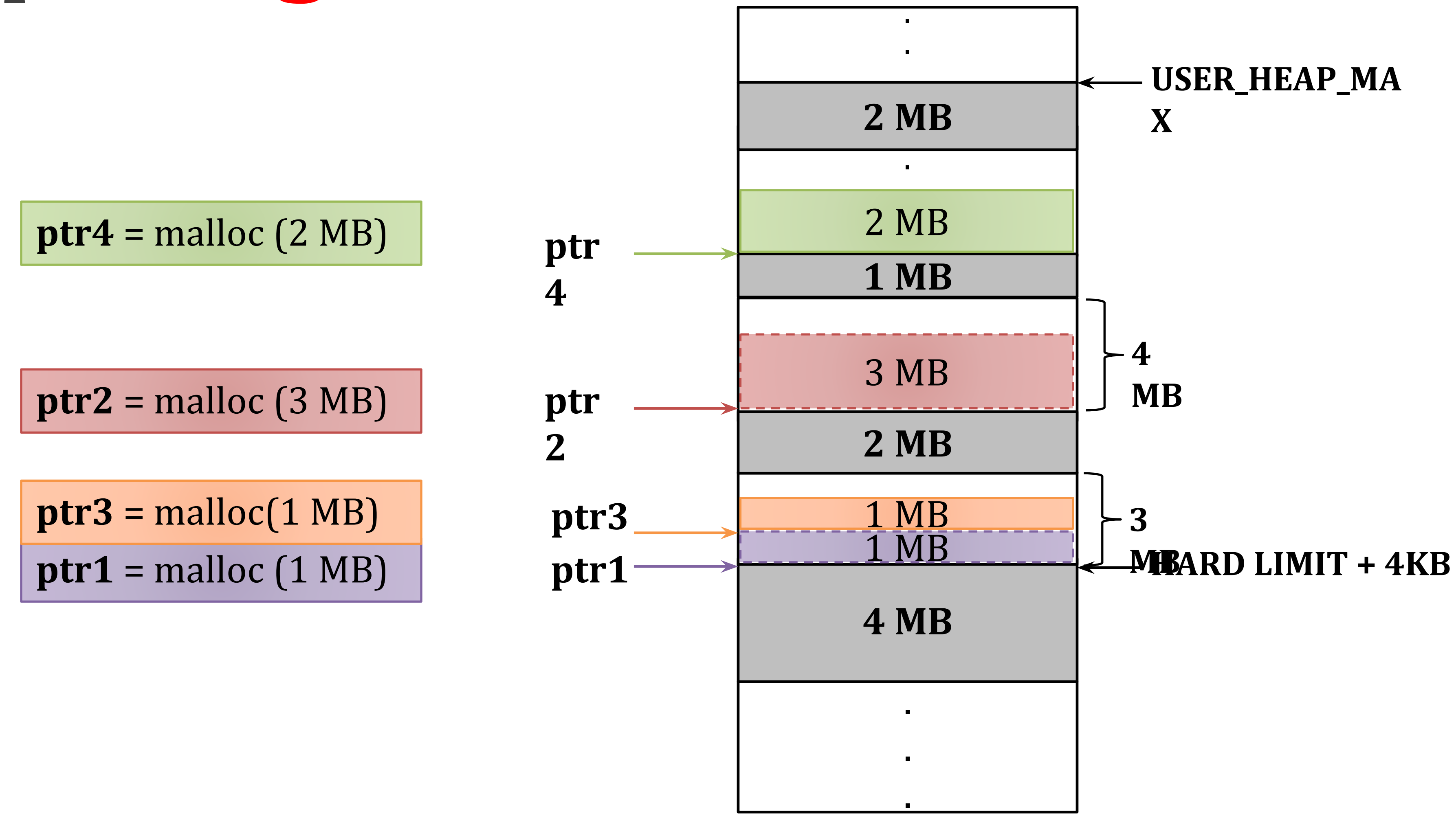
User Heap – **Page Allocator**

- **Allocate pages on 4KB granularity**



User Heap – Page Allocator

FIRST FIT Strategy



#12: malloc()

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

Description:

[USER SIDE] `lib/uheap.c`

1. If size ≤ *DYN_ALLOC_MAX_BLOCK_SIZE*: **[BLOCK ALLOCATOR]**
   - Use dynamic allocator with FIRST FIT to allocate the required space
2. Else: **[PAGE ALLOCATOR]**
   1. Implement FIRST FIT strategy to search the page allocator for suitable space to the required allocation size (space should be on 4 KB BOUNDARY)
   2. Call **`sys_allocate_user_mem()`** to **mark** the reserved space

- If failed to allocate: return NULL

To access the environment data, use `myEnv` pointer

#13: allocate_user_mem()

```
void allocate_user_mem(struct Env* e, uint32 va, uint32 size)
```

Description:

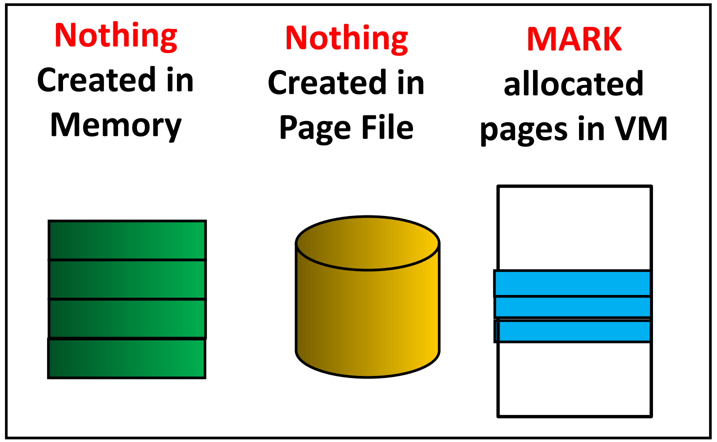
[KERNEL SIDE] `kern/mem/chunk_operations.c`:

1. **Mark** the given range to indicate it’s **reserved** for the page allocator of this environment
2. **NOTE**: you can use `create_page_table()` to create non-exiting tables (if any)

Testing:

```
FOS> run tm1 3000
```
□ **PAGE ALLOCATOR**

```
FOS> run tm2 3000
```
□ **BLOCK ALLOCATOR**



Nothing Created in Memory

Nothing Created in Page File

MARK allocated pages in VM

#14: free()

```
void free(void* virtual_address)
```

Description:

[USER SIDE] `lib/uheap.c`

To access the environment data,

use `myEnv` pointer

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
   1. **Find** the allocated size of the given virtual_address
   2. **Free** this allocation from the page allocator of the user heap
   3. Call "`sys_free_user_mem()`" to free the allocation from the memory & page file

- Else (i.e. invalid address): should `panic(…)`

#15: free_user_mem()

```
void free_user_mem(struct Env* e, uint32 va, uint32 size)
```

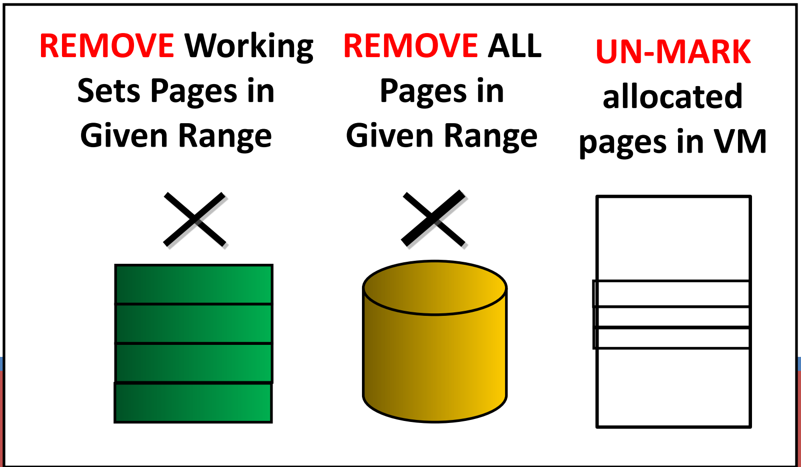
Description:

[KERNEL SIDE] `kern/mem/chunk_operations.c`:

1. **Unmark** the given range to indicate it’s **NOT reserved** for the page allocator of this environment
2. **Free ALL pages** of the given range from the **Page File** (Check MS2 appendix for PAGE FILE)
3. **Free** ONLY pages that are resident in the **working set** from the memory (Check MS2 appendix for WS)

Testing:

```
FOS> run tf1 3000  □ free in PAGE ALLOCATOR
FOS> run tf2 3000  □ free in BLOCK ALLOCATOR
FOS> run tff1 3000 □ first fit PAGE ALLOCATOR
FOS> run tff2 3000 □ first fit BLOCK ALLOCATOR
```



REMOVE Working Sets Pages in Given Range

REMOVE ALL Pages in Given Range

UN-MARK allocated pages in VM

BONUS#3: O(1) of free_user_mem

```
void free_user_mem(struct Env* e, uint32 va, uint32 size)
```

Description:

- Efficient **O(1)** implementation of removing page from WS List **instead** of **searching** the entire list

Testing:

- **[UNSEEN] test at your own**

User Heap: **Testing**

- Test each function in MS2 independently in a **FRESH SEPARATE RUN**.

- The time limit of each individual test: **max of 10 sec / each**

#	Test Functionality	Test
1	***tst_malloc_1.c (tm1):*** tests **malloc()** & **allocate_user_mem()** in **PAGE ALLOCATOR**. It validates: 1. return addresses from the malloc() 2. NOTHING is allocated in page file or memory 3. memory access (read & write) of the allocated spaces (placement of fault handler should work) 4. number of allocated frames and the WS entries after each memory access	□ **FOS>** run tm1 3000
2	***tst_malloc_2.c (tm2):*** tests **malloc() & sys_sbrk()** in **DYNAMIC ALLOCATOR**. It validates: 1. return addresses from the malloc() 2. NOTHING is allocated in page file 3. memory access (read & write) of the allocated spaces (placement of fault handler should work) 4. number of allocated frames and the WS entries after each memory access	□ **FOS>** run tm2 3000

`"test [TEST NAME] completed. Evaluation = …%”`

To ensure the success of a test, this message like this **MUST be appeared without any ERROR messages or PANICs**.

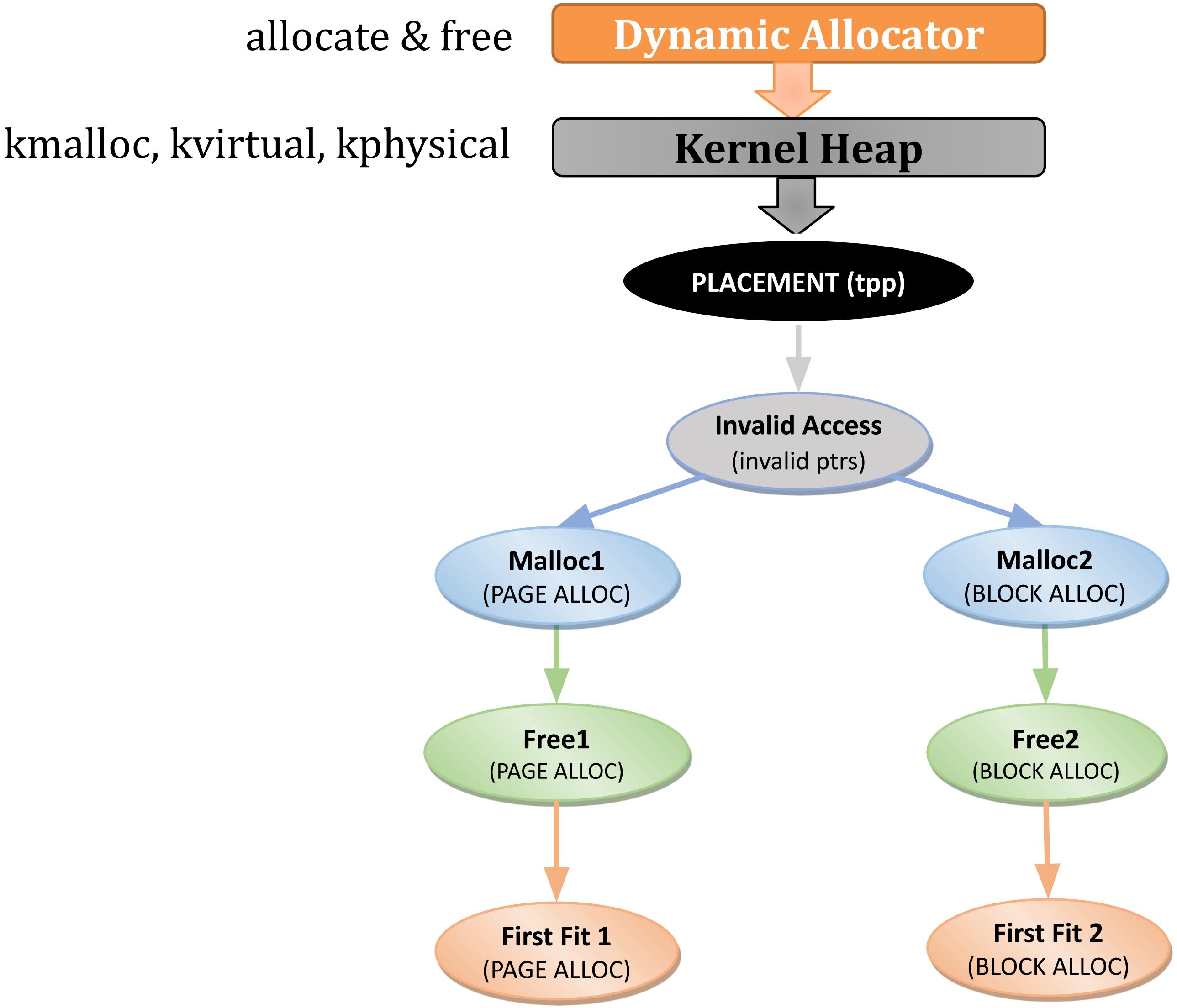
User Heap: **Testing**

□ Test each function in MS2 independently in a **FRESH SEPARATE RUN**.

□ The time limit of each individual test: **max of 10 sec / each**

#	Test Functionality	Test
3	***tst_free_1.c (tf1):*** tests the implementation **free()** & **free_user_mem()** in **PAGE ALLOCATOR**. It validates: 1. number of freed frames by free_user_mem() 2. Removing the allocated pages from working set (if any) 3. memory access (read & write) of the removed spaces (should not be allowed)	□ **FOS>** run tf1 3000
4	***tst_free_2.c (tf2):*** tests the implementation **free()** in **DYNAMIC ALLOCATOR**. It validates: 1. Coalesce (merge) cases after free 2. Allocate after free in merged blocks 3. number of freed frames (should not be affected) 4. allocated pages in working set (should not be affected)	□ **FOS>** run tf2 3000
5	***tst_first_fit_1.c (tff1):*** tests the **FIRST FIT strategy** in **PAGE ALLOCATOR**. Tests both granted and non-granted requests. (It depends on `free & free_user_mem`).	□ **FOS>** run tff1 3000
6	***tst_first_fit_2.c (tff2):*** tests the **FIRST FIT strategy** in **DYNAMIC ALLOCATOR**. Tests both granted and non-granted requests. (It depends on `free`).	□ **FOS>** run tff2 **10,000**

User Heap: **Testing Dependency Graph**



Agenda

- Logistics
- Part 0: Code Updates
- **Part 1: Kernel Heap**
  - Block Allocator
  - Page Allocator
- **Part 2: Fault Handler I**
- **Part 3: User Heap**
  - Block Allocator
  - Page Allocator
- **Part 4: Shared Memory**
- Summary & Quick Guide
- How to submit?

Shared Memory

The main functions required to handle “**Shared Memory**” are:

#	Function	File
1	`smalloc (User side)`	**Functions definitions TO DO in: lib/uheap.c**
2	`sget (User side)`	

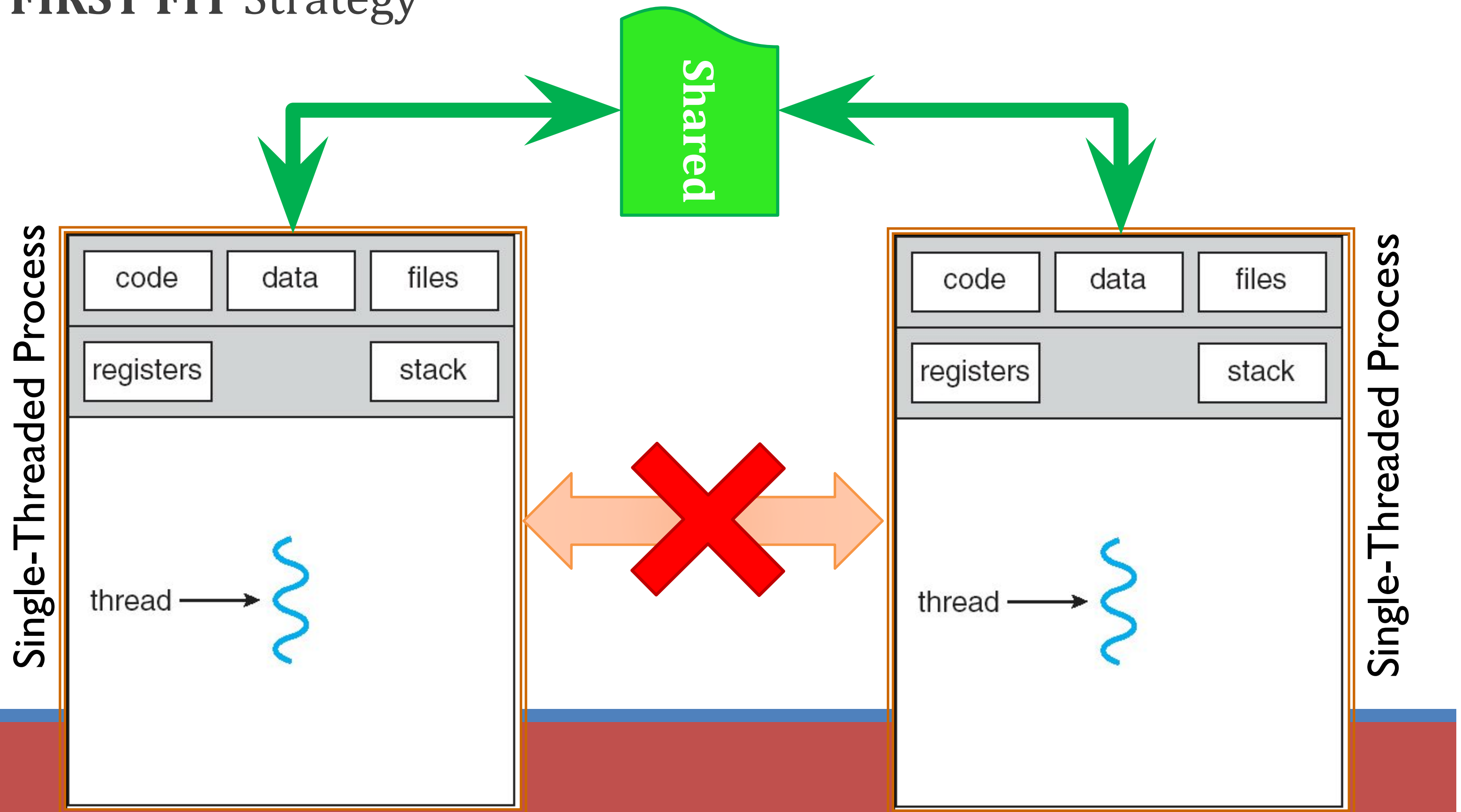
3	`create_share(),`
	`create_frames_storage()`
		**Functions definitions TO DO in:
		kern/mem/shared_memory_manager.c**

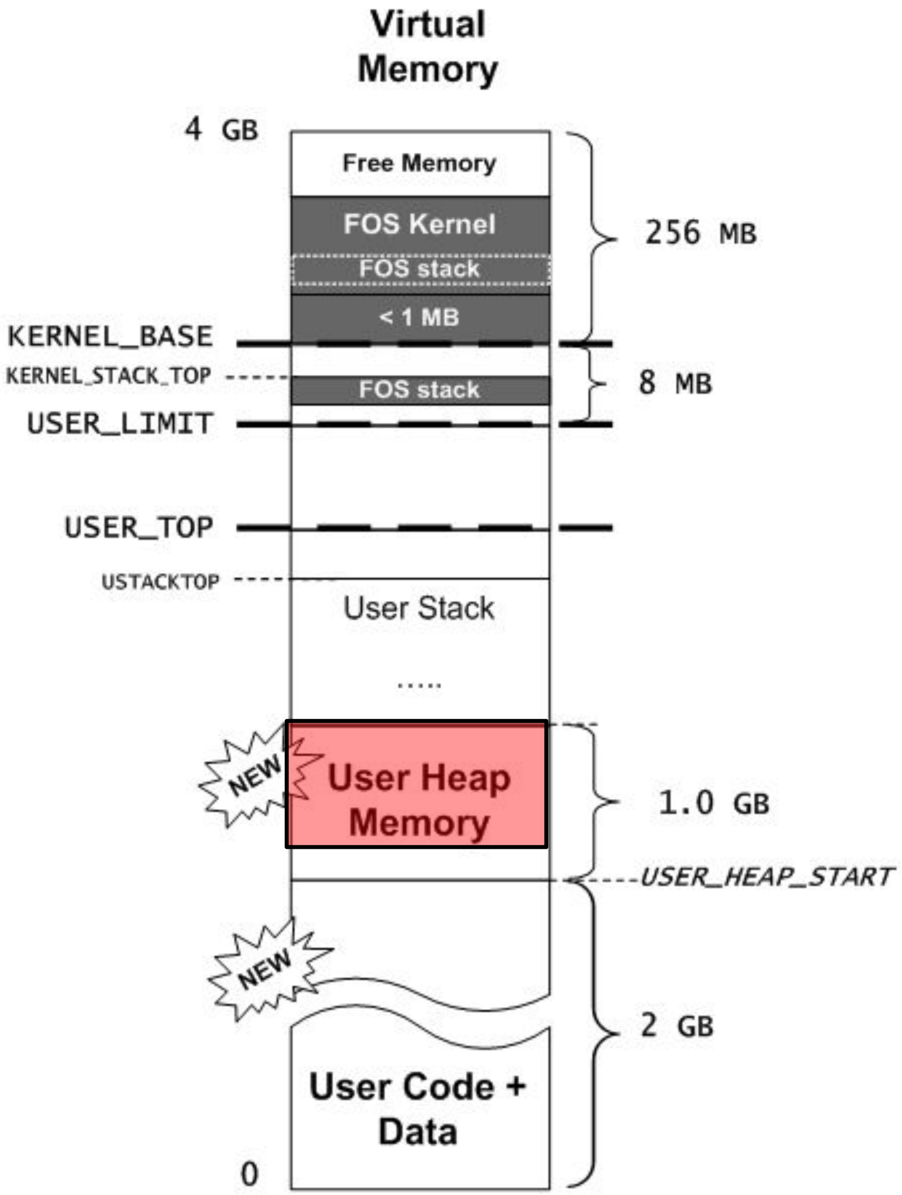
4	`get_share`	
5	`createSharedObject(Kernel side)`	
6	`getSharedObject(Kernel side)`	

MS2
BONUS 4
	`sfree (User side)`
	`free_share()` & `freeSharedObject()`
	`(Kernel side)`
		**lib/uheap.c
		kern/mem/shared_memory_manager.c**

Shared Memory: **Overview**

- Communication is **harder** between processes
- To allow it: **shared memory** is applied
  - Create and share objects in the **PAGE ALLOCATOR** of USER HEAP
  - **FIRST FIT** Strategy





Shared Memory: **Overview**

Creation (Application 1):

```
int* ptr_sharedInt;

uint8 isWritable = 1;

ptr_sharedInt = smalloc(“mySharedInt”,4,isWritable);
```

- allocate 4 bytes named “mySharedInt” in virtual memory and return the allocated virtual address to “ptr_sharedInt”
- Specify its shared permission to be **writable**

```
*ptr_sharedInt = 70;
```

- Set the value of the shared int to 70

Shared Memory: **Overview**

Access from other app. (Application 2):

```
int* ptr_sharedInt;

ptr_sharedInt = sget(App1ID, “mySharedInt”);
```

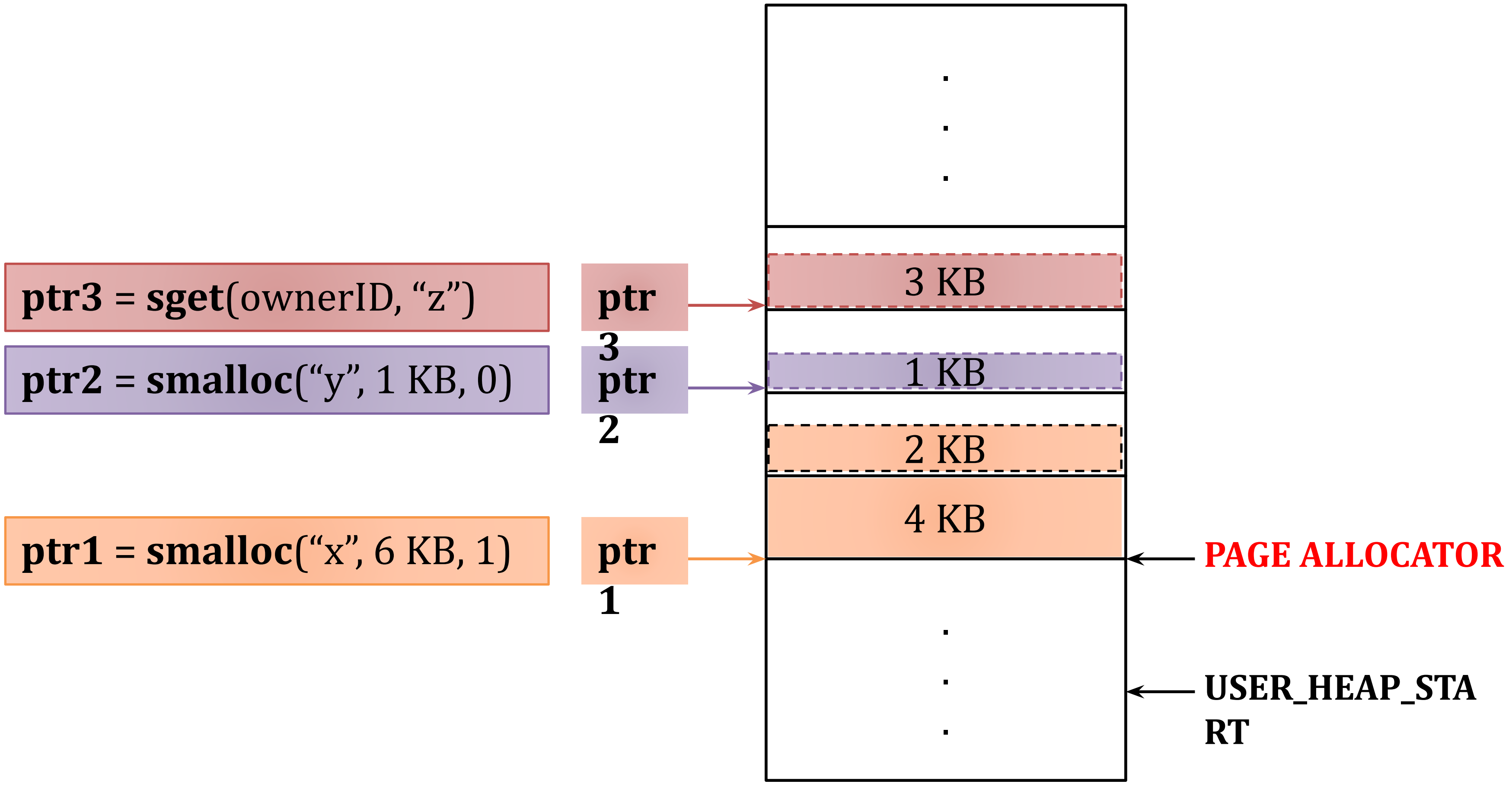
- Search for the shared object, named “mySharedInt” and belong to App1ID
- share it in app2, and return its virtual address in “ptr_sharedInt”

```
int sharedInt = *ptr_sharedInt;
```

- Read its value (it should be 70)

Shared Memory: **Overview**

Create/Share pages on 4KB granularity


ptr3 = **sget**(ownerID, “z”)
ptr2 = **smalloc**(“y”, 1 KB, 0)
ptr1 = **smalloc**(“x”, 6 KB, 1)
ptr 3
ptr 2
ptr 1
3 KB
1 KB
2 KB
4 KB
PAGE ALLOCATOR
USER_HEAP_START

Shared Memory: **Overview**

FIRST FIT Strategy

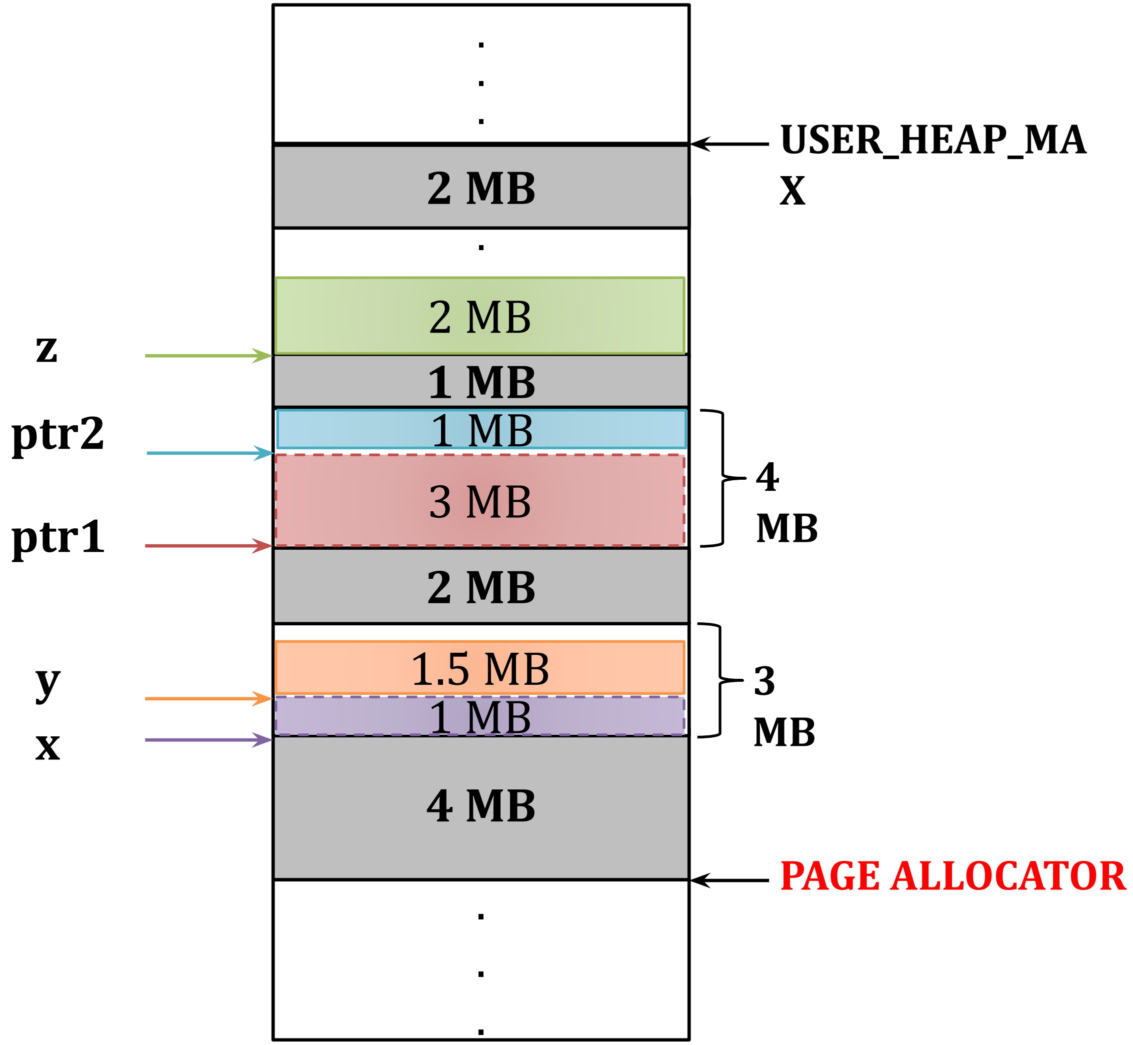
z = **sget**(ownerID, “z”)

ptr3 = malloc (1 MB)

ptr2 = malloc (3 MB)

y = **smalloc**(“y”,1.5MB,0)

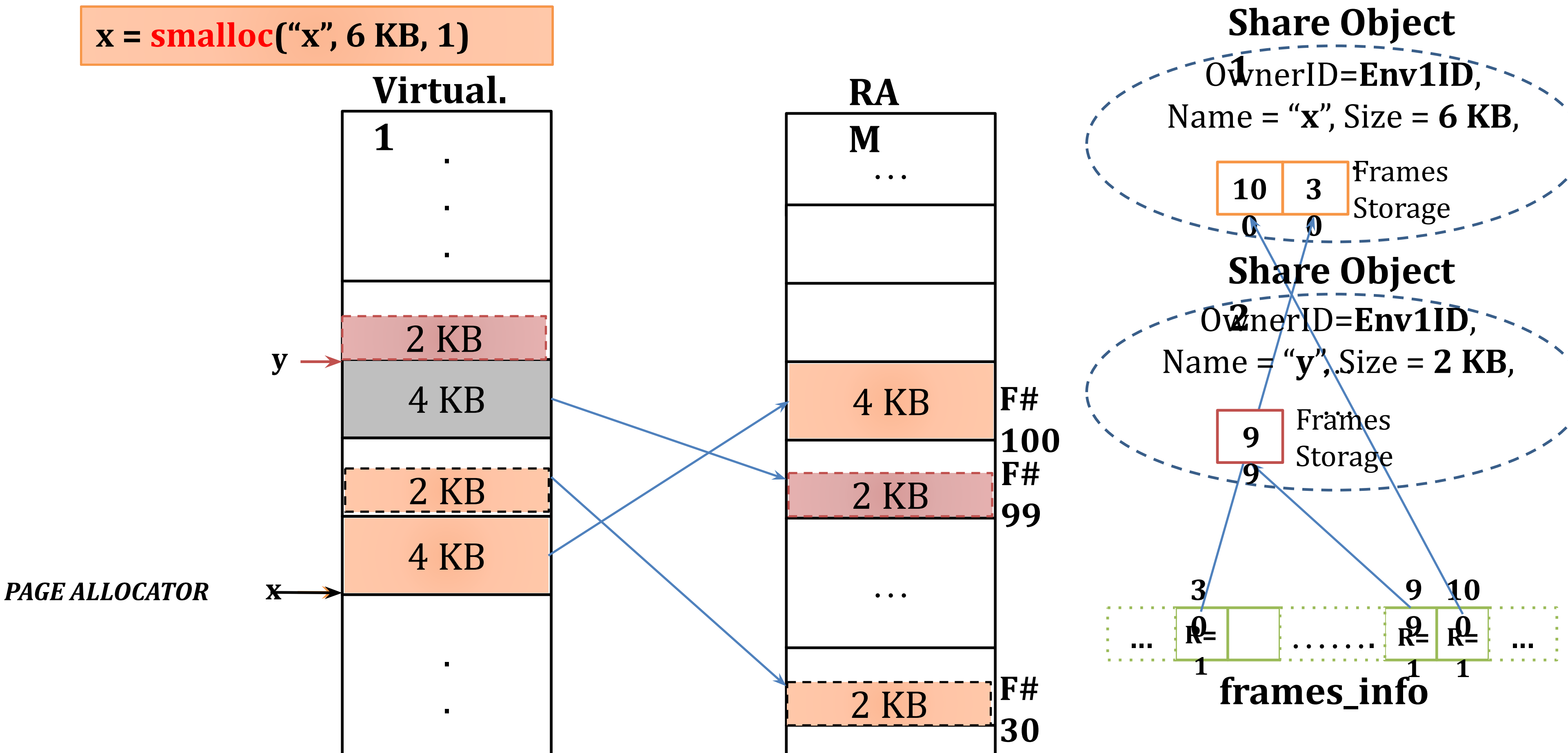
x = **smalloc**(“x”, 1 MB, 1)



Shared Memory: **Details**

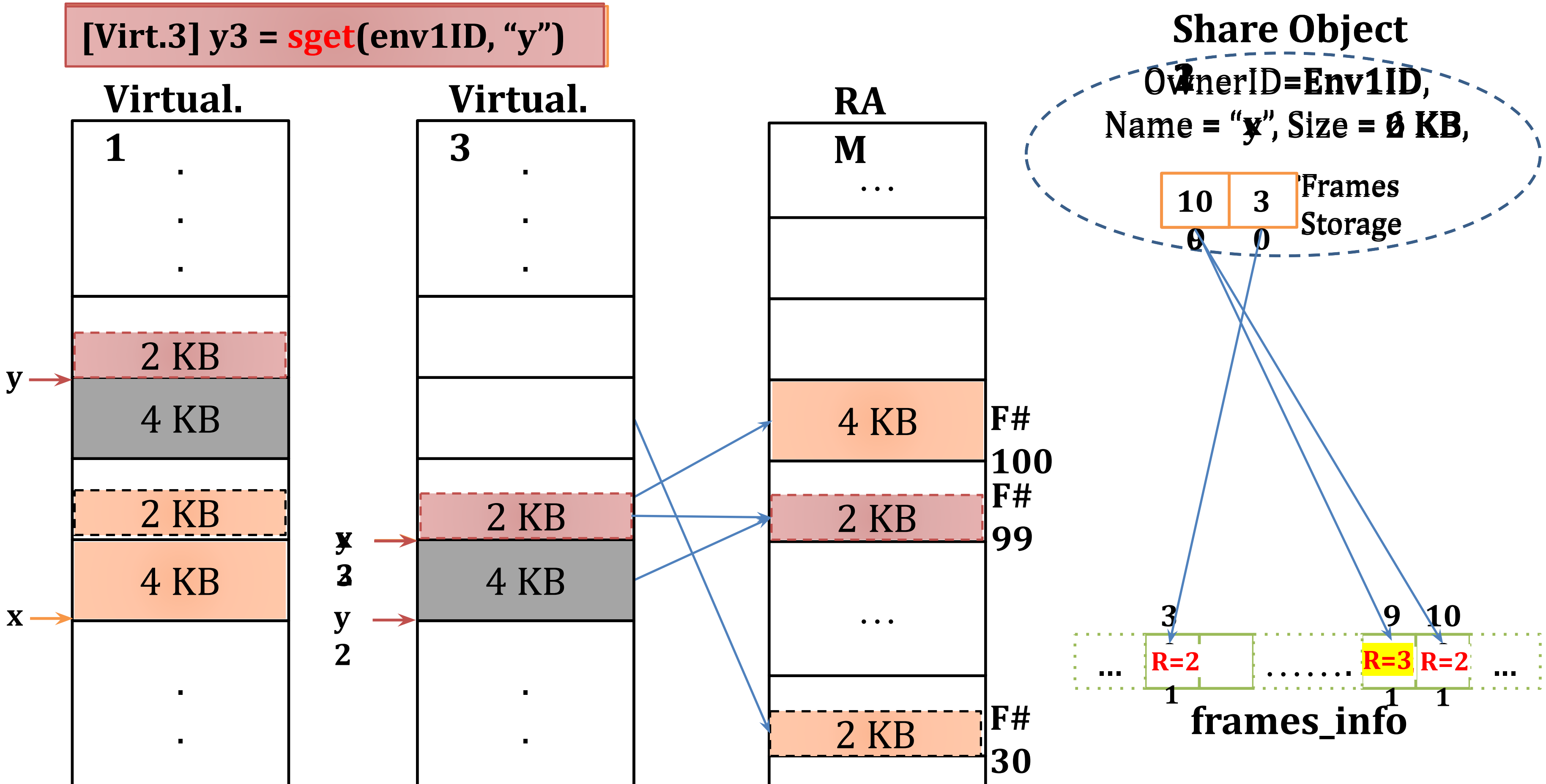
1. Space MUST be allocated in RAM
2. KEEP track of the allocated frames

smalloc(): Store allocated frames for later use



Shared Memory: **Details**

sget(): Share the stored frame of the object.



Shared Memory: **Data [GIVEN]**

kern/mem/shared_memory_manager.h

```
struct Share
{
    //Unique ID for this Share object
    //Should be set to VA of created object after masking most significant bit (to make it +ve)
    int32 ID ;
    char name[64];        //share name
    int32 ownerID ;       //ID of the owner environment

    int size;             //share size
    uint32 references;    //references, number of envs looking at this shared mem object
    uint8 isWritable;     //sharing permissions (0: ReadOnly, 1:Writable)

    struct FrameInfo** framesStorage;    //to store frames to be shared

    LIST_ENTRY(Share) prev_next_info;    // list link pointers
}
```

Shared Memory: **Data [GIVEN]**

kern/mem/shared_memory_manager.h

```
//List of all shared objects
LIST_HEAD(Share_List, Share);        // Declares 'struct Share_List’

struct
{
    struct Share_List shares_list ;  //List of all share variables created by any process

    struct spinlock   shareslock;    //Use it to protect the shares_list in the kernel

} AllShares;
```

GENERAL NOTE: make sure to protect **`shares_list`** using its lock

Shared Memory: **Functions [GIVEN]**

kern/mem/shared_memory_manager.c

`void sharing_init()` (DONE)

- Initialize the shares list & its lock

ALREADY called for you ☺️

`int getSizeOfSharedObject(int32 ownerID, char* shareName)` (DONE)

- Get the size of the shared object

kern/mem/shared_memory_manager.c

#16: Alloc & Initialize Share Object

```
inline struct FrameInfo**
create_frames_storage(int numOfFrames)
```

1. **Create** an array of pointers to `struct FrameInfo` of size `numOfFrames`
2. **Initialize** it by ZEROs
3. **Return**:
   1. If succeed: pointer to the created array
   2. If failed: NULL

Testing:

- Will be tested during the other tests…

```
struct Share* create_share(int32 ownerID, char*
    shareName, uint32 size, uint8 isWritable)
```

1. **Allocate** a new shared object
2. **Initialize** its members:
   1. **references** = 1,
   2. **ID** = VA of created object after masking msb
3. **Create** the "**framesStorage**"
4. **Return**:
   1. If succeed: pointer to the created object for `struct Share`
   2. If failed: **UNDO** any allocation & **return** NULL

Testing:

- Will be tested during the other tests…

kern/mem/shared_memory_manager.c

#17: Search for Share Object

```
struct Share* get_share(int32 ownerID, char* name)
```

1. **Search** for shared object with the given “**ownerID**” & “**name**” in the “**shares_list**”

2. **Return**:
   1. If found: pointer to the **Share** object
   2. Else: NULL

Testing:

- Will be tested during the other tests…

[USER SIDE] `lib/uheap.c`

#18: smalloc()

```
void* smalloc(char *sharedVarName, uint32 size, uint8 isWritable)
```

1. **Apply FIRST FIT** strategy to search the **PAGE ALLOCATOR** in user heap for suitable space to the required allocation size (on **4 KB BOUNDARY**)
2. **if no suitable space** found, return NULL
3. **Call `sys_createSharedObject`**(…) to invoke the Kernel for allocation of shared variable

RETURN:

1. If successful, return its virtual address
2. Else, return NULL

Testing:

`FOS> run tshr1 3000` □ **smalloc**

`FOS> run tshr3 3000` □ **smalloc (special cases)**

[KERNEL SIDE] **kern/mem/shared_memory_manager.c**

#19: createSharedObject()

```
int createSharedObject(int32 ownerID, char* shareName, uint32 size,
            uint8 isWritable, void* virtual_address)
```

1. **Allocate & Initialize** a new share object
2. **Add** it to the “**shares_list**”
3. **Allocate ALL** required space in the **physical memory** on a PAGE boundary
4. **Map** them on the given "**virtual_address**" on the current process with **WRITABLE** permissions
5. **Add** each allocated frame to "**frames_storage**" of this shared object to keep track of them for later use

RETURN:

1. ID of the shared object (its VA after masking out its msb) if **success**
2. E_SHARED_MEM_EXISTS if the shared object **already exists**
3. E_NO_SHARE if **failed to create** a shared object

[USER SIDE] `lib/uheap.c`

#20: sget()

```
void* sget(int32 ownerEnvID, char *sharedVarName)
```

1. **Get** the size of the shared variable (use **`sys_getSizeOfSharedObject`**())
2. **If not exists**, return NULL
3. **Apply FIRST FIT** strategy to search the heap for suitable space (on 4 KB BOUNDARY)
4. **if no suitable space** found, return NULL
5. **Call `sys_getSharedObject`**(…) to invoke the Kernel for sharing this variable

RETURN:

1. If successful, return its virtual address
2. Else, return NULL

Testing:

`FOS> run tshr2 3000` □ **smalloc & sget**

`FOS> run tff3 3000` □ **First Fit (smalloc, sget, malloc & free)**

[KERNEL SIDE] `kern/mem/shared_memory_manager.c`

#21: getSharedObject()

```
int getSharedObject(int32 ownerID, char* shareName, void* virtual_address)
```

1. **Get** the shared object from the "**shares_list**"
2. **Get** its physical frames from the “**frames_storage**”
3. **Share** these frames with the current process starting from the given "**virtual_address**"
4. **Use** the flag **isWritable** to make the sharing either **read-only** OR **writable**
5. **Update** references

RETURN:

1. ID of the shared object (its VA after masking out its msb) if **success**
2. `E_SHARED_MEM_NOT_EXISTS` if the shared object **is NOT exists**

[USER SIDE] `lib/uheap.c`

BONUS#4: Delete Shared Object

```
void sfree(void* virtual_address)
```

1. **Find** the ID of the shared variable at the given address
2. **Call** `sys_freeSharedObject()` to free it

Testing:

`FOS> run tshr4 3000` □ **smalloc & sfree**

`FOS> run tshr5 3000` □ **smalloc , sget & sfree**

[KERNEL SIDE] kern/mem/shared_memory_manager.c

BONUS#4: Delete Shared Object

```
void free_share(struct Share* ptrShare)
```

1. **Delete** the give **share object** from the "**shares_list**"
2. **Delete** the "**framesStorage**" and the **shared object** itself

[KERNEL SIDE] `kern/mem/shared_memory_manager.c`

BONUS#4: Delete Shared Object

```
int freeSharedObject(int32 sharedObjectID, void *startVA)
```

1. **Get** the shared object from the "**shares_list**"
2. **Unmap** it from the current process
3. **If page table(s)** become **empty**, **remove** it
4. **Update** references
5. **If this is the last share**, delete the share object (use **`free_share`**())
6. **Flush** the cache

"test [TEST NAME] completed. Evaluation = …%”
To ensure the success of a test, this message like this **MUST be appeared without any ERROR messages or PANICs**.

Shared Memory: **Testing**

- Test each function in MS2 independently in a **FRESH SEPARATE RUN**.
- The time limit of each individual test: **max of 30 sec / each**

#	Function	Test
1	**`smalloc & createSharedObject`** ***tst_sharing_1.c (tshr1):*** It tests the creation of shared objects. It validates the returned addresses and the number of allocated frames. It also checks the memory access (read & write) of the created shared objects.	**FOS>** run tshr1 3000
2	**`smalloc & createSharedObject`** ***tst_sharing_3.c (tshr3):*** It tests handling the **special cases** of shared objects creation. Namely, creating objects with same name, creating large object that exceeds heap area and creating large number of objects that exceed the max allowed objects.	**FOS>** run tshr3 3000
3	**`smalloc, createSharedObject, sget & getSharedObject`** ***tst_sharing_2master.c (tshr2):*** It tests the request for sharing object. It validates the returned addresses & the number of allocated frames. It also checks the mem. access (read & write) of the retrieved shared objects with different read/write permissions.	**FOS>** run tshr2 3000
4	**`smalloc, createSharedObject, sget & getSharedObject, malloc & free`** ***tst_first_fit_3.c (tff3):*** tests the **first fit strategy** by requesting **normal and shared** allocations that always fit in one of the free segments. All requests should be granted.	**FOS>** run tff3 3000

Shared Memory: **BONUS Testing**

Test	Test Functionality
FOS> run tshr4 3000	***tst_sharing_4.c***: Tests the free of shared object after creating it.
FOS> run tshr5 3000	***tst_sharing_5_master.c:*** Tests the free of shared object after creating & getting it.

Agenda

- Logistics
- Part 0: Code Updates
- **Part 1: Kernel Heap**
  - Block Allocator
  - Page Allocator
- **Part 2: Fault Handler I**
- **Part 3: User Heap**
  - Block Allocator
  - Page Allocator
- **Part 4: Shared Memory**
- **Summary & Quick Guide**
- How to submit?

Summary

Module	Function	Diff	Testing	Files
Kernel Heap	Initialization	**L1**	Will be tested during the other tests…	kern/mem/kheap.h & kern/mem/kheap.c
	sbrk()	**L2**	**FOS>** tst kheap FF sbrk □ tests sbrk & allocate	
	kmalloc (FIRST FIT)	**L3**	1.**FOS>** tst kheap FF kmalloc 1 □ tests allocation only 2.**FOS>** tst kheap FF kmalloc 2 □ tests FF in PAGE Alloc 3.**FOS>** tst kheap FF kmalloc 3 □ tests FF in PAGE & BLK 2 & 3 depend on kfree	
	kfree	**L2**	**FOS>** FOS> tst kheap FF kfree	
	kheap_virtual_address	**L1**	**FOS>** FOS> tst kheap FF kvirtaddr	
	kheap_physical_address	**L1**	**FOS>** FOS> tst kheap FF kphysaddr	
	(+) krealloc()	**(L3)**	**UNSEEN** – Test at your own	
	(+) Fast Page Allocator	**(L3)**	**FOS>** tst kheap FF fast **(should run in < 5 sec)**	

"test [TEST NAME] completed. Evaluation = …%"

To ensure the test success, this message like this **MUST appear without any ERROR messages or PANICs**.

Summary

Module	Function	Diff	Testing	Files
Fault Handler	Kernel Dyn. Alloc for a Process	L1	Already tested in Placement test	kern/mem/working_set_manager.c kern/proc/user_enviornment.c
	Check Invalid Pointers	L1	**FOS>** run tia 15	Kern/trap/fault_handler.c
	Page_fault_handler	L2	**FOS>** run tpp 20	Kern/trap/fault_handler.c

"test [TEST NAME] completed. Evaluation = …%”

To ensure the test success, this message like this **MUST appear without any ERROR messages or PANICs**.

Summary

Module	Function	Diff.	Testing	Files
User Heap	Initialization	L1	Will be tested during the other tests	inc/environment_definitions.h kern/proc/user_environment.c

	sys_sbrk()	L2	**UNSEEN** – Test at your own	**[USER SIDE]**
				`lib/uheap.c`
				[KERNEL SIDE]
				`kern/mem/chunk_operations.c`

	malloc (FIRST FIT)	L3	1.**FOS>** run tm1 3000□ malloc PAGE ALLOC	
	allocate_user_mem()	L1	2.**FOS>** run tm2 3000□ malloc BLOCK ALLOC	
	free	L2	3.**FOS>** run tf1 3000□ free PAGE ALLOC	
	free_user_mem()	L2	4.**FOS>** run tf2 3000□ free BLOCK ALLOC 5.**FOS>** run tff1 3000□ FF PAGE ALLOC 6.**FOS>** run tff2 **10,000**□ FF BLOCK ALLOC	
	(+) O(1) free_user_mem	(L2)	**UNSEEN** – Test at your own	

`"test [TEST NAME] completed. Evaluation = …%"`

To ensure the test success, this message like this **MUST appear without any ERROR messages or PANICs**.

Summary

Module	Function	Diff.	Testing	Files

Shared Memory	Alloc & Initialize Share Object	L2	Will be tested during the other tests	[USER SIDE]
				`lib/uheap.c`
				[KERNEL SIDE]
				`kern/mem/shared_memory_manager.c`

	Search for Share Object	L1	Will be tested during the other tests	
	smalloc() (FIRST FIT)	L2	1.**FOS>** run tshr1 3000 □ smalloc	
	createSharedObject()	L2	2.**FOS>** run tshr2 3000 □ smalloc & sget	
	sget() (FIRST FIT)	L2	3.**FOS>** run tshr3 3000 □ smalloc (special cases)	
	getSharedObject()	L1	4.**FOS>** run tff3 3000 □ FF: smalloc, sget, malloc, free	
	(+) Delete Shared Object (sfree(), free_share(), freeSharedObject()	(L3)	1.**FOS>** run tshr4 3000 □ smalloc, sfree 2.**FOS>** run tshr5 3000 □ smalloc, sget, sfree	

SUMMARY: L1 □ 9 FUNCTIONS - L2 □ 10 FUNCTIONS - L3 □ 2 FUNCTIONS

"test [TEST NAME] completed. Evaluation = …%"

To ensure the test success, this message like this **MUST appear without any ERROR messages or PANICs**.

REMEMBER:

- UPDATE YOUR CODE ACCORDING TO PREVIOUSLY DESCRIBED STEPS
- READ ATATCHED APPENDICES FOR HELPER FUNCTIONS.

DEBUGGING:

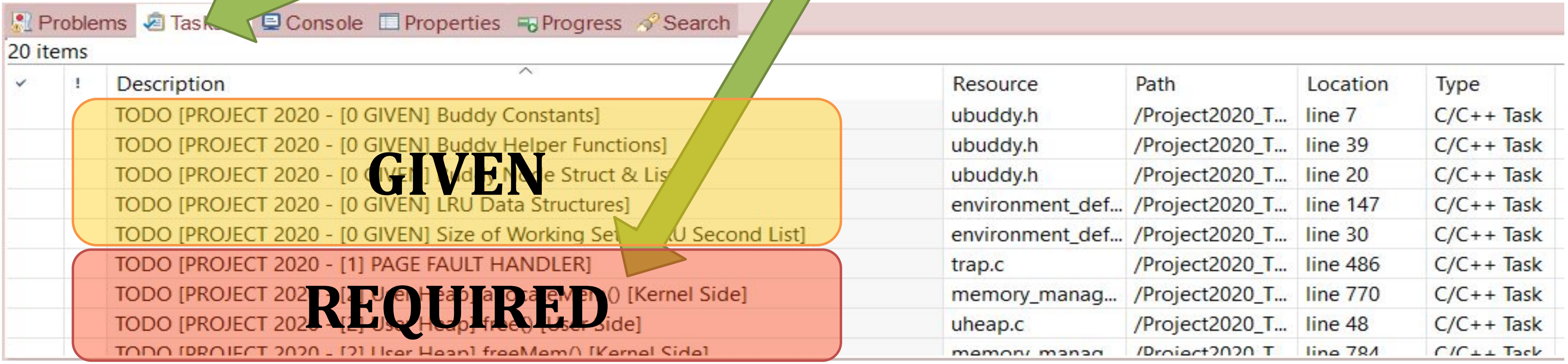
1. Debug via breakpoints (ECLIPSE) [link]
2. Debug via printing [link: 1st minute]
3. Locate the line causing exception via **disassembly** [link]

Where should I write the Code?

There’re shortcut links that direct you to the function definition

[1] Click on “Tasks” Tab

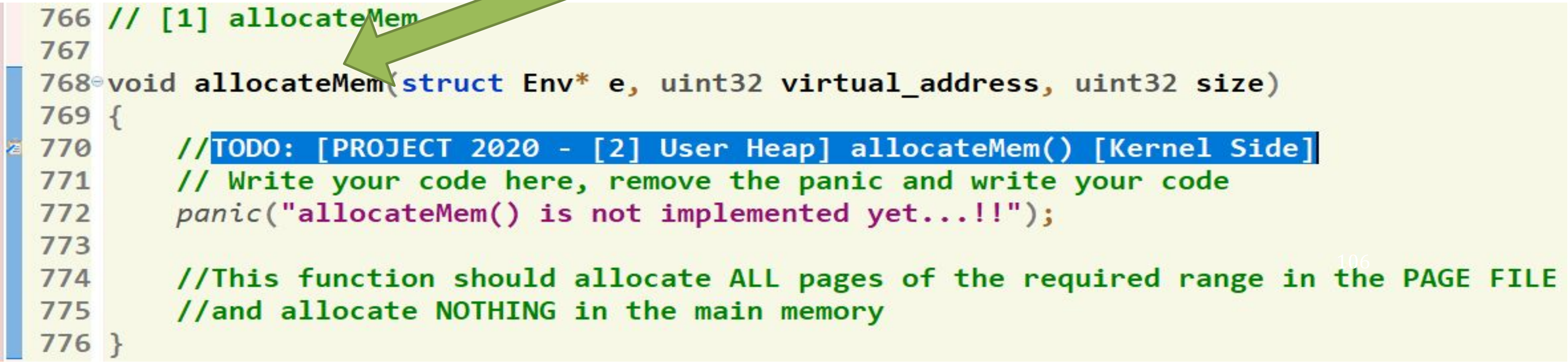
[2] Double Click on the required function



20 items

✓	!	Description	Resource	Path	Location	Type
		TODO [PROJECT 2020 - [0 GIVEN] Buddy Constants]	ubuddy.h	/Project2020_T...	line 7	C/C++ Task
		TODO [PROJECT 2020 - [0 GIVEN] Buddy Helper Functions]	ubuddy.h	/Project2020_T...	line 39	C/C++ Task
		TODO [PROJECT 2020 - [0 GIVEN] Buddy Node Struct & Lis	ubuddy.h	/Project2020_T...	line 20	C/C++ Task
		TODO [PROJECT 2020 - [0 GIVEN] LRU Data Structures]	environment_def...	/Project2020_T...	line 147	C/C++ Task
		TODO [PROJECT 2020 - [0 GIVEN] Size of Working Set [illegible]U Second List]	environment_def...	/Project2020_T...	line 30	C/C++ Task
		TODO [PROJECT 2020 - [1] PAGE FAULT HANDLER]	trap.c	/Project2020_T...	line 486	C/C++ Task
		TODO [PROJECT 2020 - [2] User Heap] allocateMem() [Kernel Side]	memory_manag...	/Project2020_T...	line 770	C/C++ Task
		TODO [PROJECT 2020 - [2] User Heap] free() [User Side]	uheap.c	/Project2020_T...	line 48	C/C++ Task
		TODO [PROJECT 2020 - [2] User Heap] freeMem() [Kernel Side]	memory_manag...	/Project2020_T...	line 784	C/C++ Task

[3] Function body, at which you should write the code



```
// [1] allocateMem

void allocateMem(struct Env* e, uint32 virtual_address, uint32 size)
{
	//TODO: [PROJECT 2020 - [2] User Heap] allocateMem() [Kernel Side]
	// Write your code here, remove the panic and write your code
	panic("allocateMem() is not implemented yet...!!");

	//This function should allocate ALL pages of the required range in the PAGE FILE
	//and allocate NOTHING in the main memory
}
```

Agenda

- Logistics
- Part 0: Code Updates
- **Part 1: Kernel Heap**
  - Block Allocator
  - Page Allocator
- **Part 2: Fault Handler I**
- **Part 3: User Heap**
  - Block Allocator
  - Page Allocator
- **Part 4: Shared Memory**
- **Summary & Quick Guide**
- **How to submit?**

Submission Rules

Read the following instructions as the code correction is done AUTOMATICALLY. Any violation in these rules will lead to 0 and, in this case, nothing could be happened.

First ensure the following that (READ CAREFULLY):

- You tested each function in a **FRESH RUN** and a congratulations message have been appeared.
- **NO CODE with errors WILL BE CORRECTED**. So, CLEAN & RUN your project several times before your submission.
- You submitted **BEFORE** the deadline by several hours to **AVOID** any internet problems.
- **DEADLINE: THU of Week #9 (28/11 @11:59 PM)**
- **NO DELAYED submissions WILL BE ACCEPTED**.
- **ONLY ONE person** from the team shall submit the code.
- The **TEAM # MUST BE CORRECT**.
- **DON'T take the FORM LINK FROM ANYONE**. OPEN the form from its LINK ONLY. **Otherwise, your submission is AUTOMATICALLY CANCELLED by GOOGLE**.
- You **MUST RECEIVE A MAIL FROM GOOGLE with your submission after clicking submit**. If **nothing received, re-submit again to consider your submission**.

Submission Steps

STEPS to SUBMIT:

- Step 1: Clean & run your code the last time to ensure that there are any errors.
- Step 2: Create a new folder and name it by your team number **ONLY**. Example 1 or 95. [**ANY extra chars will lead to 0**].
- Step 3: **DELETE** the "obj" folder from the "FOS_PROJECT_2024_Template"
- Step 4: PASTE the "FOS_PROJECT_2024_Template" in the folder created in step #2.
- Step 5: Zip the created new folder. Its name shall be like **[num of your team.zip]**. [**ANY extra chars will lead to ZERO**].
- Step 6: Open the form from **HERE.**
- Step 7: Fill your team's info .. Any wrong information will cancel your submission, revise them well.
- Step 8: Upload the zipped folder in step 5 to the form in its field.
- Step 9: MUST RECEIVE A MAIL from GOOGLE with your submission, otherwise re-submit again.

Thank you for your care…

Enjoy **making** your **own FOS** ☺