OS’25 Project

PART III: **INDIVIDUAL MODULES**

(FAULT HANDLER II, USER HEAP, SHARED MEMORY, CPU SCHED, KERN PROTECTION)

Agenda

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

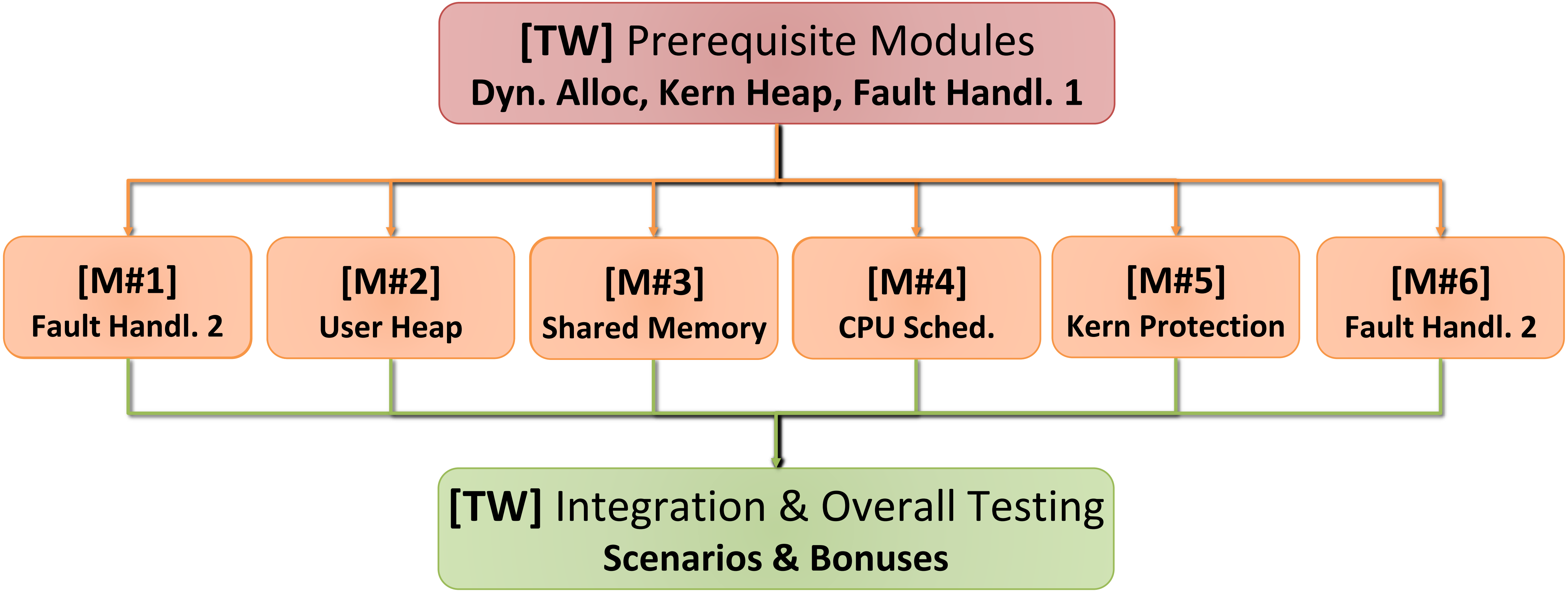
Agenda

- **PART II: INDIVIDUAL MODULES**
  - Fault Handler II (Replacement)
  - User Heap
  - Shared Memory
  - CPU Scheduling
  - Kernel Protection

Fault Handler II (Replace)

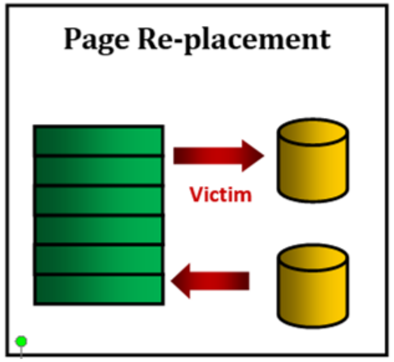
PART III: INDIVIDUAL MODULE #1

Project Overview



Objective

Handle the page fault exception by choosing a **victim page** to be **replaced** with the **faulted page**



Fault Handler II: **Replacement**

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

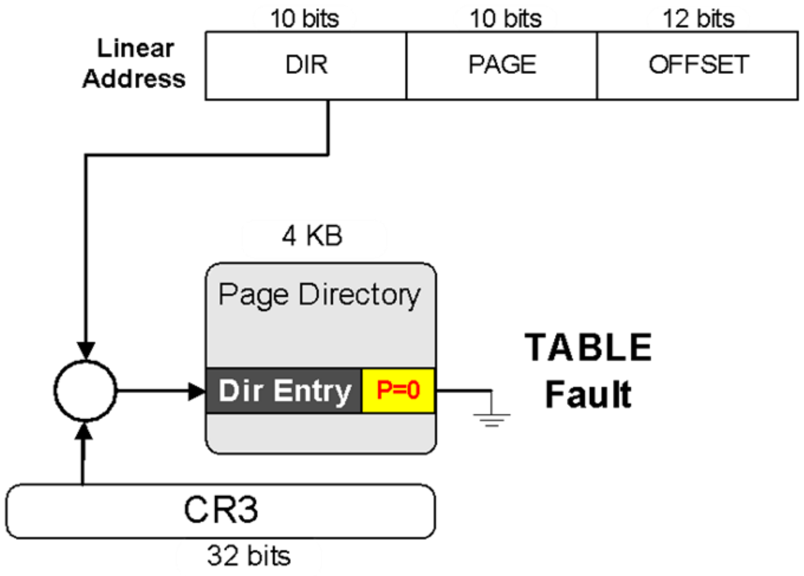
#	Function	File
1	`get_optimal_num_faults`	**Functions definitions TO DO in: kern/trap/fault_handler.c**
2	`page_fault_handler: find reference stream`	
3	`page_fault_handler: Clock Replacement`	

Fault Handler II: **Introduction**

- **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)

    <u>OR</u>
  - **A page** can't be accessed due to either

CASE1: Table not exist



Repl.: **Optimal Alg. – Idea**

- Selects page for which time to the **next reference is the longest**
- **Adv: BEST** method (benchmark for others)
- **DisAdv: Not feasible** – need perfect knowledge of future

Reference Stream	2	1	5	2	4	5	3	2	5	2

Initial WS

2
3

Repl.: **Optimal Alg. – Details**

- **Initial WS** is filled when the process is loaded into RAM (in “**env_create**”)
- How to find the **Reference Stream**?
  1. Run the process to the end **WITHOUT CHANGING** the **Initial WS (no replace)**
  2. Force invoking the OS at each page reference... **HOW?**
  3. Keep track of the referenced pages in a list
- At the end, **calculate** the number of page faults by **tracing** the **reference stream starting** from the **initial WS**

Repl.: **Optimal Alg. – Details**

- Force invoking the OS at each page reference... **HOW?**
  - **OPT1:** RESET PRESENT BIT FOR EACH PAGE

  If a single instruction needs two or more pages at same time infinite faults!!

```
void libmain(int argc, char **argv)
{
    //...

    if (printStats)
    {
        char isOPTReplCmd[100] = "__IsOPTRepl__" ;
```

Need 3 pages at same time

Ref Stream: 0, 1, 0, 14, 0, 1, 0, 14 , 0, 1, 0, 14 , 0, 1, 0, 14 , 0, 1, 0, 14...

Repl.: **Optimal Alg.** – **Details**

- Force invoking the OS at each page reference... **HOW?**
  - **OPT2:** RESET PRESENT BITS ONLY IF ACTIVE WORKING SET IS CHANGED

    If a faulted page in the Active WS, do nothing

    Else, if WS is FULL, reset present bit of its pages & delete it

Ex: if WS Max Size = 3, Initial WS = {2, 3 }

Ref Stream: 0, 1, 0, 14, ~~0, 1, 0, 14 , 0, 1, 0, 14 , 0, 1, 0, 14 , 0, 1, 0, 14...~~

Present Bit: 0 1 1 1

Repeated Pattern is Cancelled

Active WS

2
3

2	1	1	1
3		0	0
0			14

Repl.: **Optimal Alg. – Given Data**

1. Struct for **page reference** with the **prev-next pointer** (to be used as list element)

```
struct PageRefElement {
    unsigned int virtual_address;
    LIST_ENTRY(PageRefElement) prev_next_info;  // list link pointers
};
```

1. List of reference stream inside the `struct Env` (**already** initialized by `LIST_INIT()`)

```
struct PageRef_List referenceStreamList;
```

Refer to **APPENDICES** for:

Page File Helper Functions

LISTS Helper Macros

#1: Reference Stream

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

```
if(isPageReplacmentAlgorithmOPTIMAL())
{
        [1] Keep track of the Active WS
        [2] If faulted page not in memory, read it from disk
                Else, just set its present bit
        [3] If the faulted page in the Active WS, do nothing
                Else, if Active WS is FULL, reset present & delete all its
pages
        [4] Add the faulted page to the Active WS
        [5] Add faulted page to the end of the reference stream list
}
```

Refer to APPENDICES for:

Page File Helper Functions

LISTS Helper Macros

#2: Trace Num of Faults

```
int get_optimal_num_faults(struct WS_List *initWorkingSet,
    int maxWSSize, struct PageRef_List *pageReferences)
```

Description:

Calculate, by **tracing**, **# page faults** according to the **OPTIMAL** replacement strategy, Given:

1. Initial Working Set List (that the process started with)
2. Max Working Set Size
3. Page References List (contains the stream of referenced VAs till the process finished)

IMPORTANT: This function **SHOULD NOT** change any of the given lists

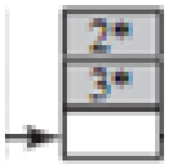
Repl.: **Clock Alg. – Idea**

- Uses the **last WS index** and an additional bit called a "**used bit**". This bit is set to 1 when the page is accessed.

1. OS scans the WS, flipping all 1's to 0
2. The first page with use bit = 0 is replaced.

Reference Stream 2 1 5 2 4 5 3 2 5 2

Initial WS



IMP. NOTE: you should maintain `page_last_WS_element` & **correct FIFO order** of the List in rest of code (`page_fault_handler`, `free_user_mem` ...)

```
//...
//=================
/*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

CLK

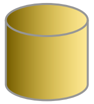
Proc Limit

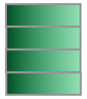
inc/environment_definitions.h

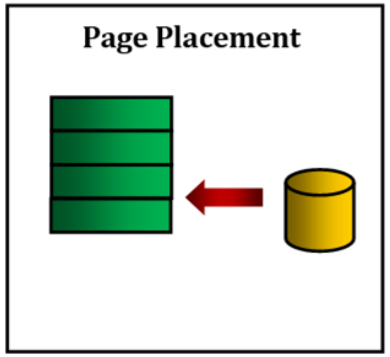
```
struct WorkingSetElement {
    unsigned int virtual_address;
    unsigned int time_stamp ;
    unsigned int sweeps_counter;
    LIST_ENTRY(WorkingSetElement) prev_next_info;
```

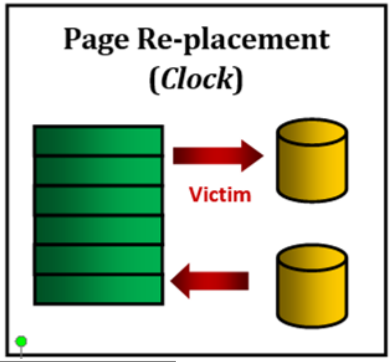
- Each process has a **working set LIST** that is initialized in **env_create()**
- Its **max size** is set in "**page_WS_max_size**" during the **env_create()**
- "**page_last_WS_element**" will point to either:
  - the **next location** in the WS after the last set one If **list is full**.
  - **Null** if the list is **NOT full**.

- This list hold pointers to **struct** containing info about the currently loaded pages in memory.
- Each struct holds important values about each page:
  1. User virtual address of the page

Refer to APPENDICES for:  Page File Helper Functions

 Working Set Structure & Helper Functions





#3: CLOCK Re/placement

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

```
if(isPageReplacmentAlgorithmCLOCK())
{
    if the size of page_WS_list < its max size, then do
    {
        Scenario 1: Placement
        //[DONE in GROUP MODULE]: [PROJECT'25] PAGE FAULT HANDLER – Placement
    }
    else
    {
        Scenario 2: Replacement
        //TODO: [PROJECT'25] PAGE FAULT HANDLER –CLK Replac.
    }
}
```

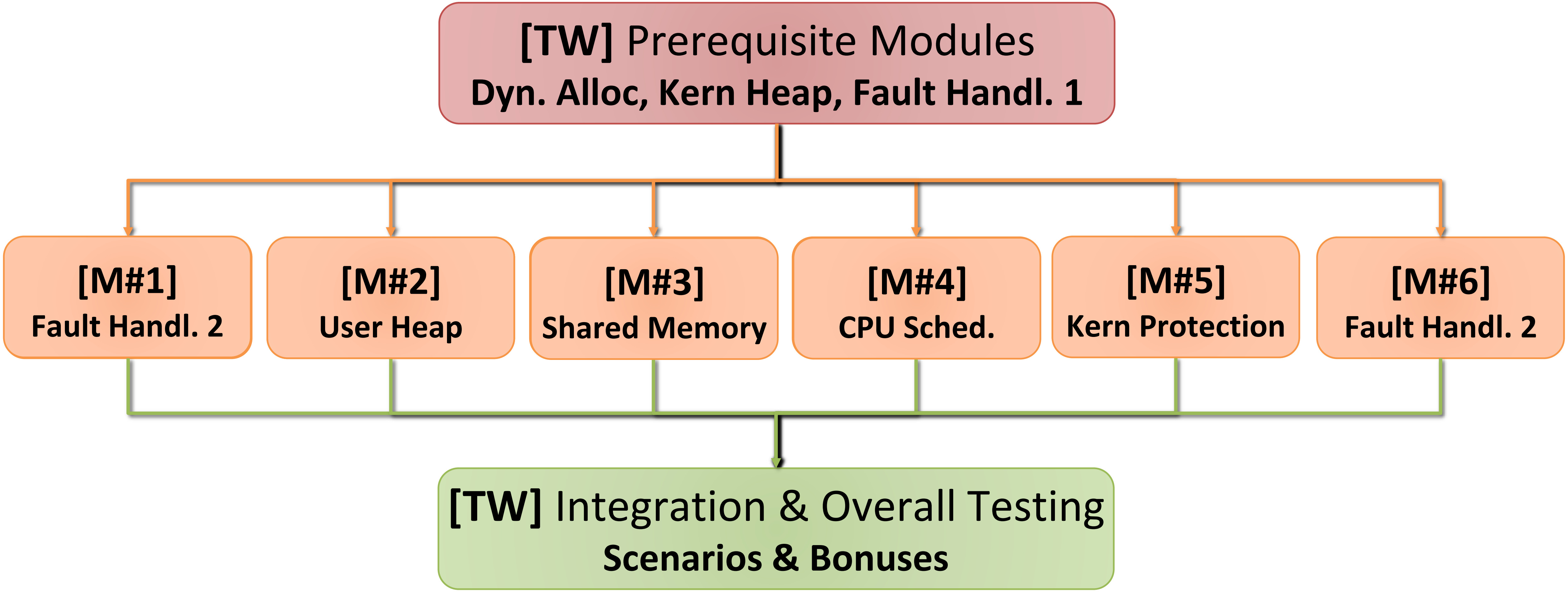
Page Fault Handler: **Switching…**

- To switch the replacement from the FOS prompt:
  - **`FOS> optimal`** → switch the replacement to OPTIMAL
  - **`FOS> clock`** → switch the replacement to CLOCK
  - To switch the replacement from the code:
- Inside the **`fault_handler_init()`** in "**`kern/trap/fault_handler.c`**":
  - `setPageReplacmentAlgorithmOPTIMAL()`
  - `setPageReplacmentAlgorithmCLOCK()`

Fault Handler II (Replace)

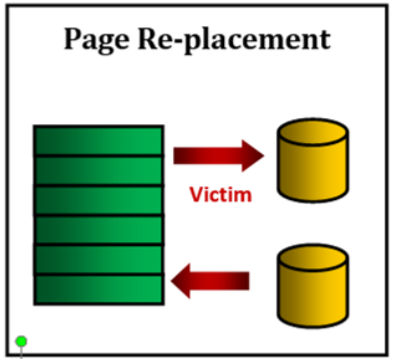
PART III: INDIVIDUAL MODULE #6

Project Overview



Objective

Handle the page fault exception by choosing a **victim page** to be **replaced** with the **faulted page**



Fault Handler II: **Replacement**

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

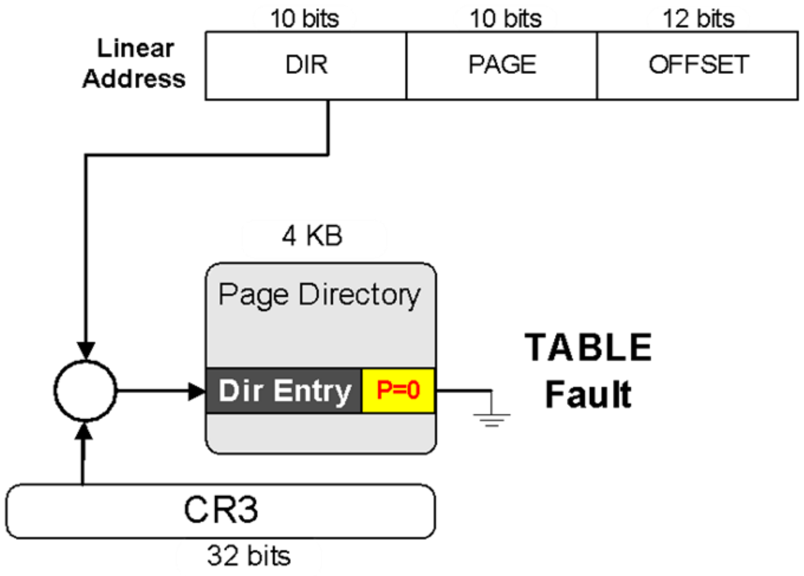
#	Function	File
1	`update_WS_time_stamps`	**Functions definitions <u>TO DO</u> in: kern/cpu/sched.c**
2	`page_fault_handler: find reference stream`	**Functions definitions <u>TO DO</u> in: kern/trap/fault_handler.c**
3	`page_fault_handler: Clock Replacement`	

Fault Handler II: **Introduction**

- **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)

    <u>**OR**</u>
  - **A page** can't be accessed due to either

CASE1: Table not exist



Repl.: **LRU Alg. – Idea**

Replaces the page that has **not been referenced for the longest time.**

Principle of locality, should be **least likely to be referenced** in the **near future**

Adv: BEST feasible

Disadv: Difficult to implement .. WHY??

- HOW do you tag each page with the **time** of last **reference to be able to implement LRU**.

Reference Stream	2	1	5	2	4	5	3	2	5	2

Initial WS

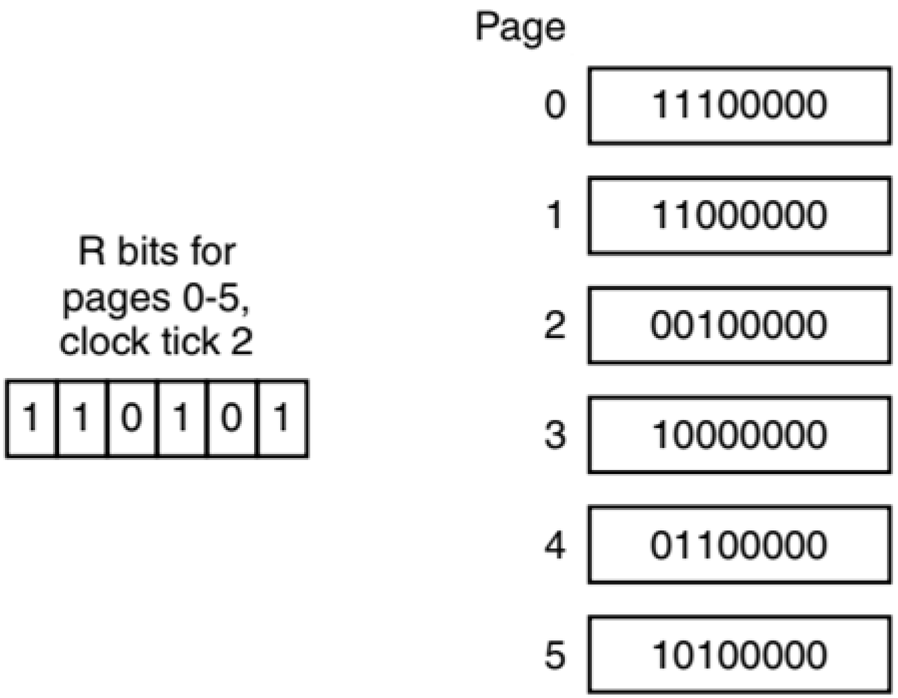
2
3

Repl.: **LRU Alg. – Details**

- HOW do you tag each page with the **time?**
  1. Maintain a 32-bit counter for the **AGE** of each page
  2. At each clock interrupt, the counter of each page in WS is **shifted right 1 bit**
  3. For each page in WS, "**Used Bit**" is **added** to the **leftmost** and **reset to 0**
- When a page fault occurs, the page with **lowest counter** is removed.
- **Intuition:**
  - page **not been referenced** for **K** clock ticks has **K** leading zeros in its counter have a **lower value** than a counter that has not been referenced for **K-1** clock ticks.

Repl.: **LRU Alg.** – **Details**

- HOW do you tag each page with the **time?**



Repl.: **LRU Alg. – Given Data**

- Counter inside the Working Set Element

inc/environment_definitions.h

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

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

Entry Manipulation In Tables

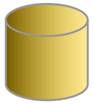
#1: Update Counters

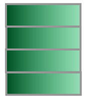
```
void update_WS_time_stamps()
```

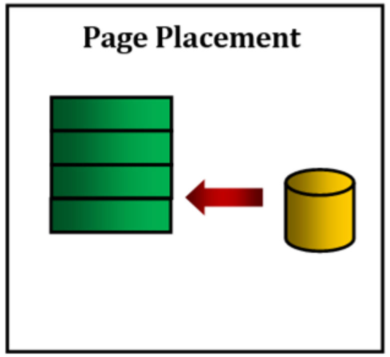
kern/cpu/sched.c

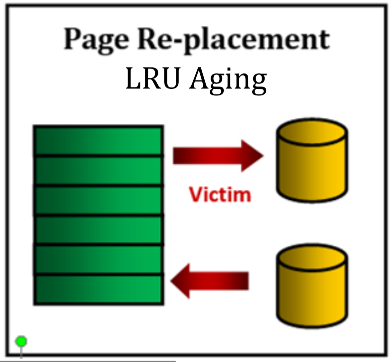
Description:

- Automatically called at every clock tick
- For each page in the WS:
  1. **Shift** its counter one-bit to the right
  2. **Add** its “Used Bit” to the leftmost bit of the counter
  3. **Clear** the “Used Bit”

Refer to **APPENDICES** for:  Page File Helper Functions

 Working Set Structure & Helper Functions





#2: LRU Re/placement

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

```
if(isPageReplacmentAlgorithmLRU(PG_REP_LRU_TIME_APPROX))
{
    if the size of page_WS_list < its max size, then do
    {
        Scenario 1: Placement

        //[DONE in GROUP MODULE]: [PROJECT'25] PAGE FAULT HANDLER – Placement
    }
    else
    {
        Scenario 2: Replacement

        //TODO: [PROJECT'25] PAGE FAULT HANDLER –LRU Replac.
    }
}
```

Repl.: **Modified Clock Alg.** – **Idea**

- Uses the **last WS index,** “**Used Bit**” and “**Modified Bit**”
- 4 states: (u, m)
  1. Not accessed recently, not modified (0, 0)
  2. Accessed recently, not modified (1, 0)
  3. Not accessed recently, modified (0, 1)
  4. Accessed recently, modified (1, 1)
- **BEST** candidate: (0, 0)… **WHY?**

Repl.: **Modified Clock Alg. – Idea**

- **Try 1*: (search for a "not used, not modified")***
  - Search for used bit = 0 and modified bit = 0
  - If found, **Replace it**, set pointer to next page
  - If not found after 1 complete cycle, goto **Try 2**
- **Try 2: *(normal clock)***
  - Search for used bit = 0, and setting the used bit value of any page in the way to 0
  - If found, **Replace it**, set pointer to next page
  - If not found after 1 complete cycle, goto **Try 1**

P#^{used,mod}

Init WS	3 r	1 r	3 w	6 r	4 w	3 r	5 r	3 w	7 w
7^{11}									
3^{10}									
?									

IMP. NOTE: you should maintain `page_last_WS_element` & **correct FIFO order** of the List in rest of code (`page_fault_handler`, `free_user_mem` ...)

```
//...
//=================
/*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

Modified CLK

Proc Limit

inc/environment_definitions.h

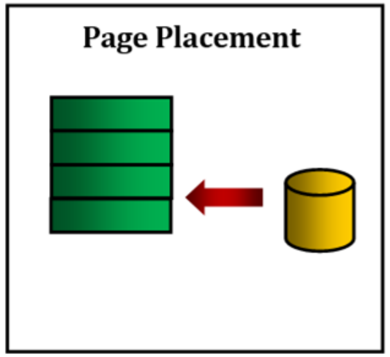
```
struct WorkingSetElement {
    unsigned int virtual_address;
    unsigned int time_stamp ;
    unsigned int sweeps_counter;
    LIST_ENTRY(WorkingSetElement) prev_next_info;
```

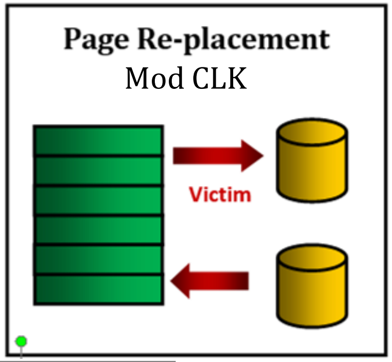
- Each process has a **working set LIST** that is initialized in **`env_create`**()
- Its **max size** is set in "**`page_WS_max_size`**" during the **`env_create`**()
- "**`page_last_WS_element`**" will point to either:
  - the **next location** in the WS after the last set one If **list is full**.
  - **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 important values about each page:
  1. User virtual address of the page

Refer to <u>APPENDICES</u> for: Page File Helper Functions

Working Set Structure & Helper Functions





#3: Modified CLK Re/place.

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

```
if(isPageReplacmentAlgorithmModifiedCLOCK())
{
    if the size of page_WS_list < its max size, then do
    {
        Scenario 1: Placement
        //[DONE in GROUP MODULE]: [PROJECT'25] PAGE FAULT HANDLER – Placement
    }
    else
    {
        Scenario 2: Replacement
        //TODO: [PROJECT'25] PAGE FAULT HANDLER – Modified CLK Replac.
    }
}
```

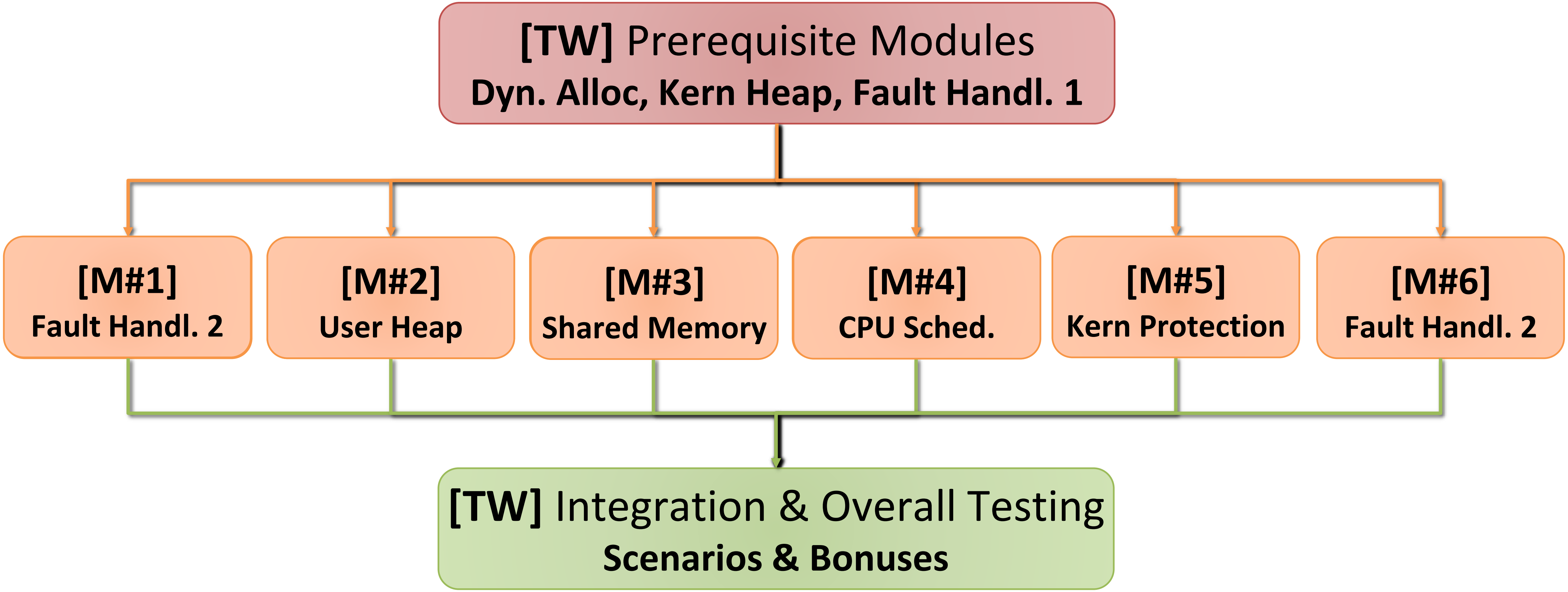
Page Fault Handler: **Switching…**

- To switch the replacement from the FOS prompt:
  - **`FOS> lru`** switch the replacement to AGING LRU
  - **`FOS> modclock`** switch the replacement to MODIFIED CLOCK
  - To switch the replacement from the code:
- Inside the **`fault_handler_init()`** in "**`kern/trap/fault_handler.c`**":
  - **`setPageReplacmentAlgorithmLRU(PG_REP_LRU_TIME_APPROX)`** set AGING LRU
  - **`setPageReplacmentAlgorithmModifiedCLOCK()`**

User Heap

PART III: INDIVIDUAL MODULE #2

Project Overview



Objective

Allow the user process to **dynamically** **allocate/free** data of **different sizes** in run-time

User Heap

IMPORTANT NOTE

TESTING depend on the Implementation of Page Fault **PLACEMENT**

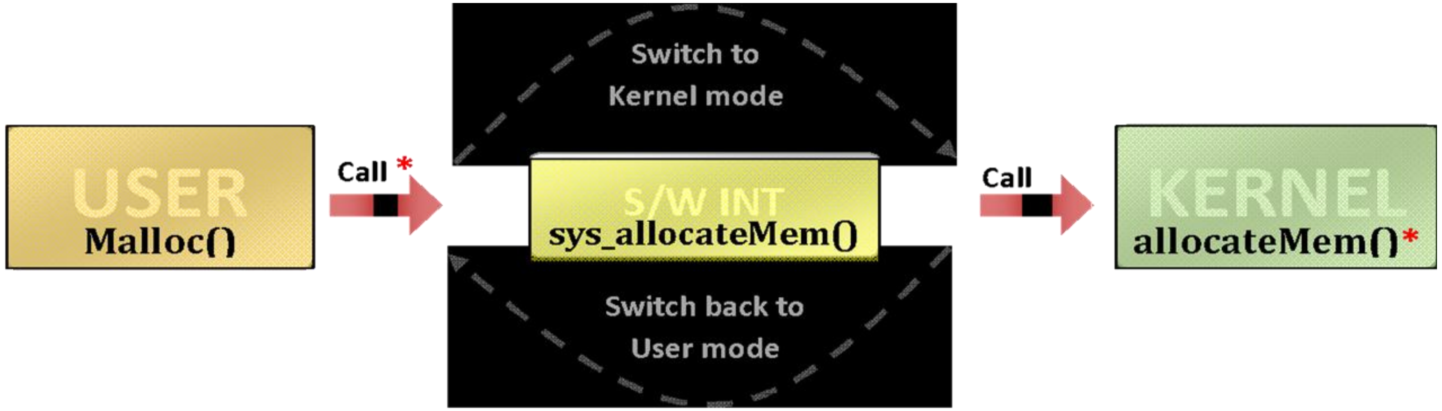
User Heap – **Functions**

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

#	Function	File
1	`malloc() (CUSTOM FIT) [USER SIDE]`	declarations: inc/uheap.h **definitions: lib/uheap.c**
2	`free() [USER SIDE]`	
3	`allocate_user_mem [KERNEL SIDE]`	**Kern/mem/chunk_operations.c**
4	`free_user_mem [KERNEL SIDE]`	

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

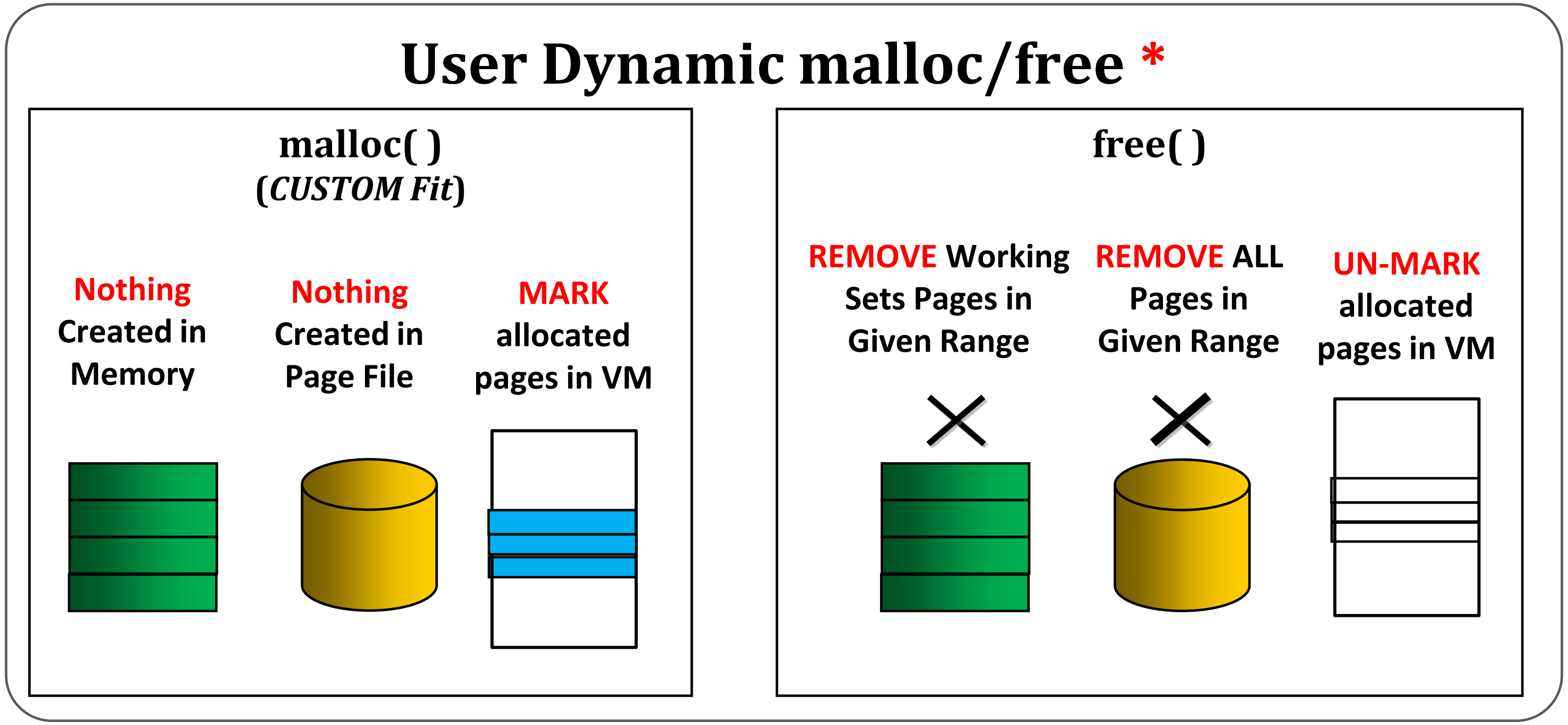
User Heap: **Overview**

- **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**

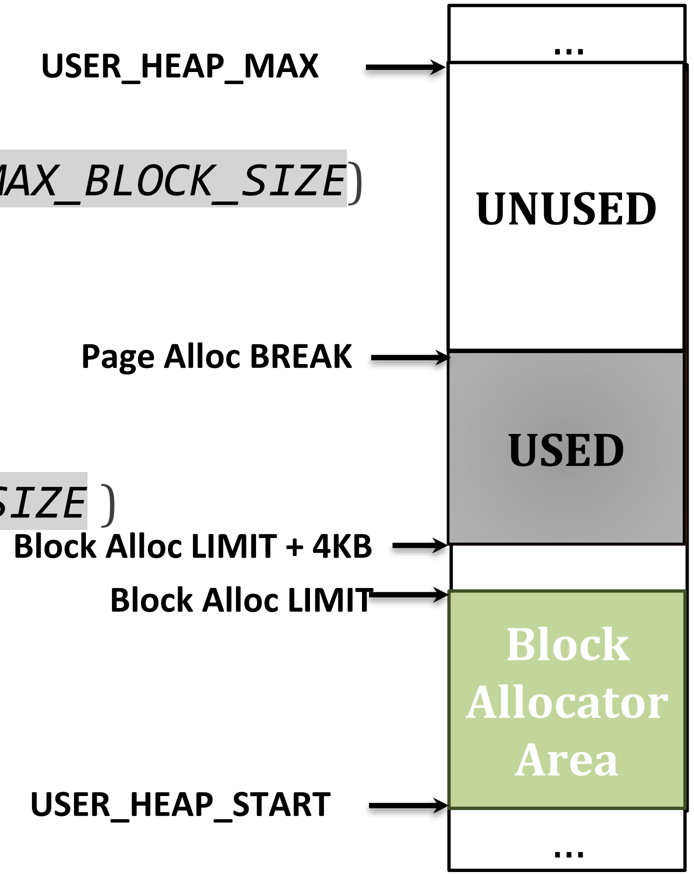


USE PERM_UHPAGE TO UN/MARK PAGES

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 Functions
   3. Range: [USER_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, USER_HEAP_MAX)
      1. **USED** Area: [BLK ALLOC LIMIT + PAGE_SIZE, PAGE ALLOC BREAK)
      2. **UNUSED** Area: [PAGE ALLOC BREAK , USER_HEAP_MAX)



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**

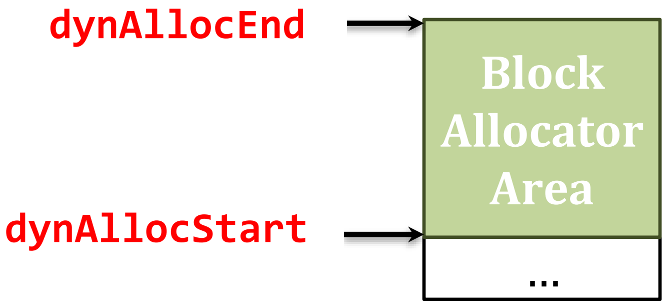
1. Has 2 limits:
    1. `dynAllocStart`: begin of block allocator area
    2. `dynAllocEnd`: end of block allocator area

2. Use Dynamic Allocator with its data structure

3. **Already initialized**, together with the dynamic allocator itself inside:

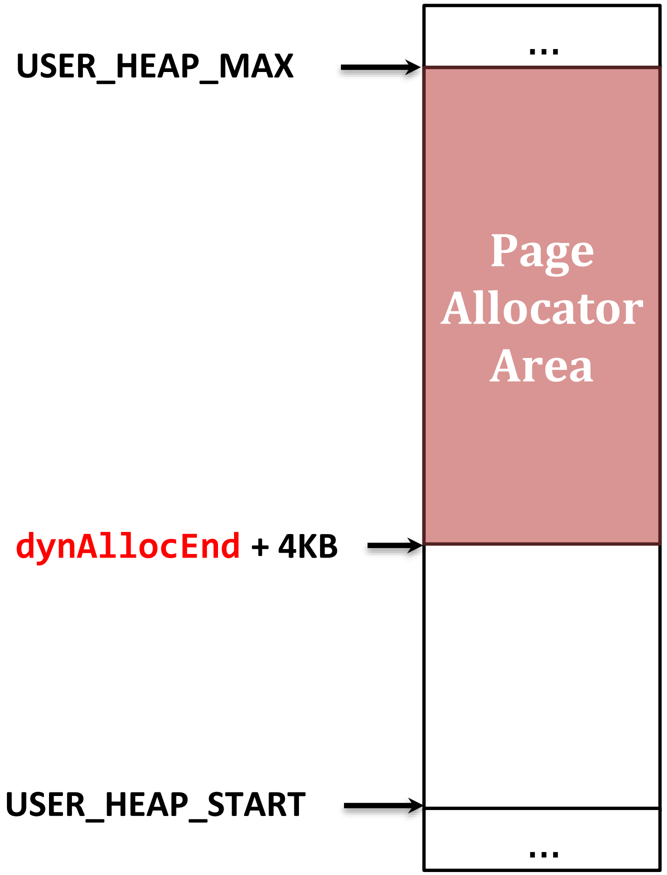
    `int uheap_init(…)` defined in **`lib/uheap.c`**

- This function, in turn, is **already called** in **`malloc`**()



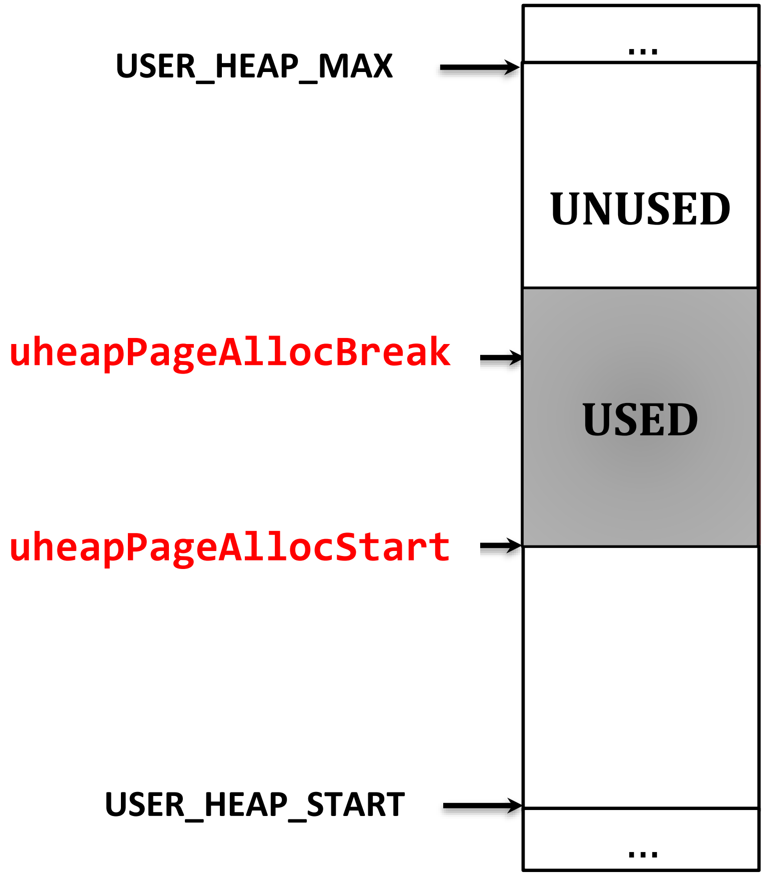
User Heap – Page Allocator

- 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: **CUSTOM FIT**



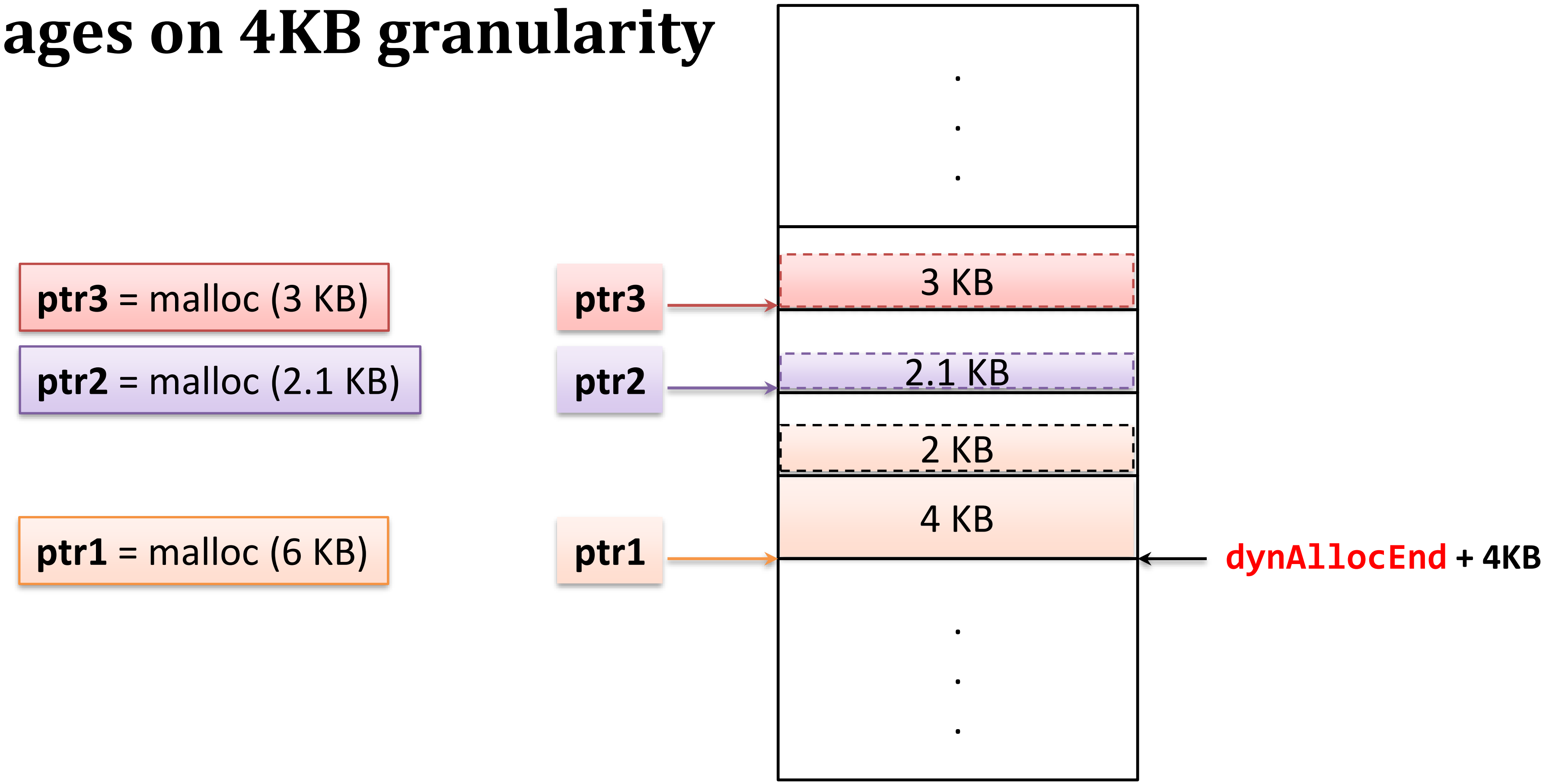
User Heap – Page Allocator

1. Has 2 limits defined in **`inc/uheap.h`**:
   1. `uheapPageAllocStart`: begin of page allocator area
   2. `uheapPageAllocBreak`: end of currently used area
2. malloc/free can move `uheapPageAllocBreak` up/down



User Heap – Page Allocator (malloc)

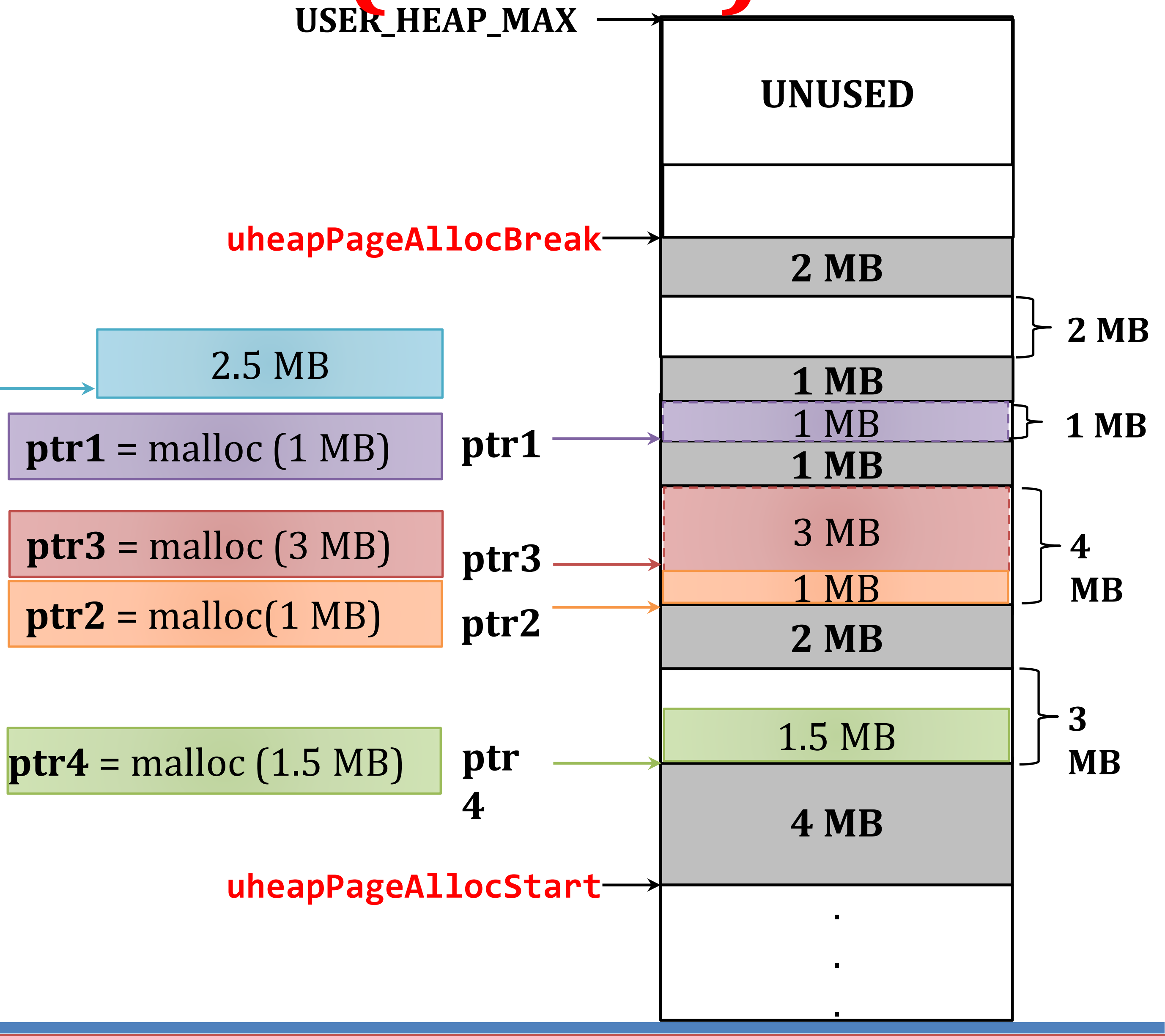
Allocate pages on 4KB granularity



User Heap – Page Allocator (malloc)

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**

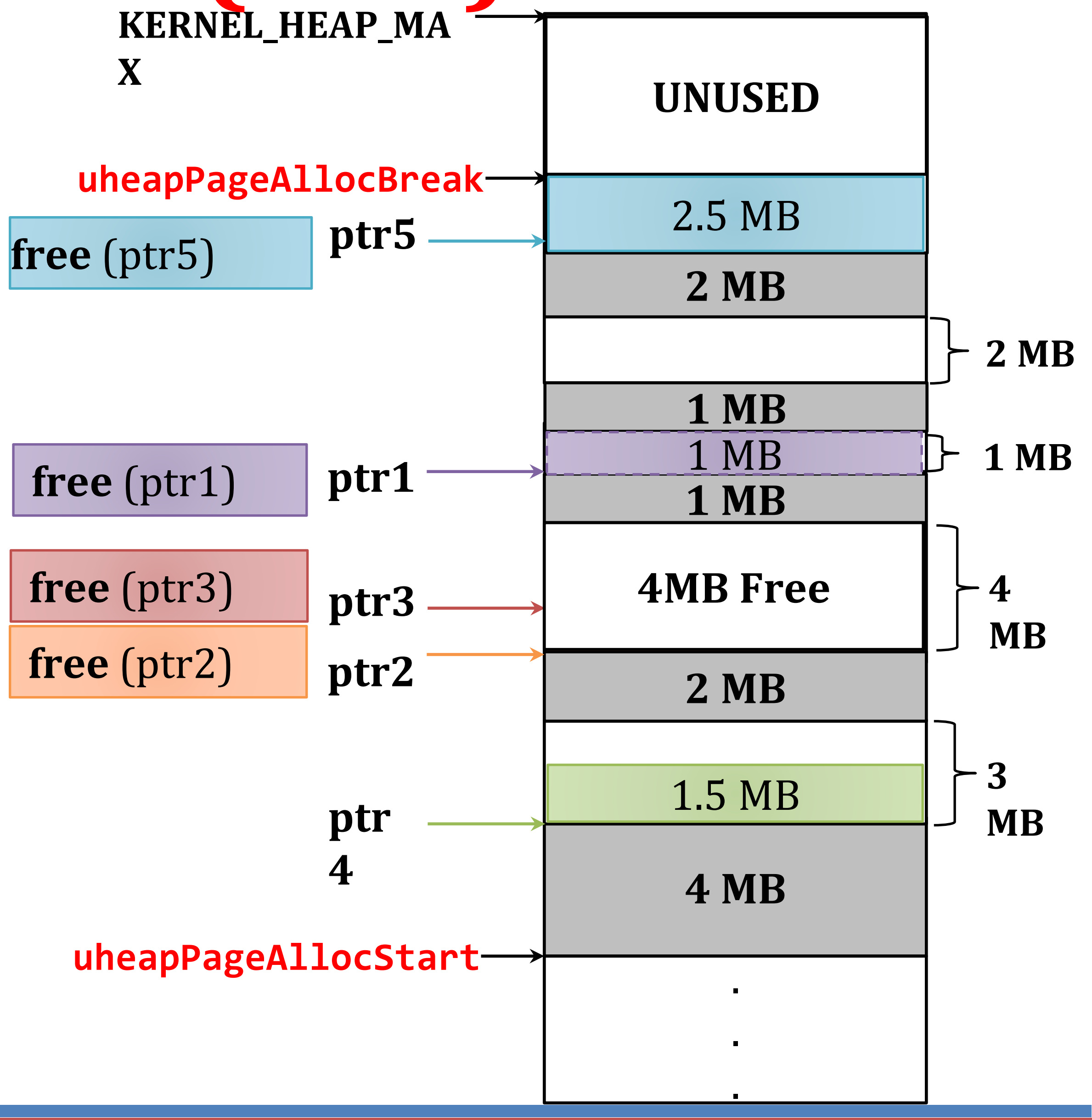


User Heap – Page Allocator (free)

Make sure to:

1. Merge adjacent blocks

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



#1: 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 to allocate the required space
2. Else: **[PAGE ALLOCATOR]**
   1. Implement CUSTOM 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

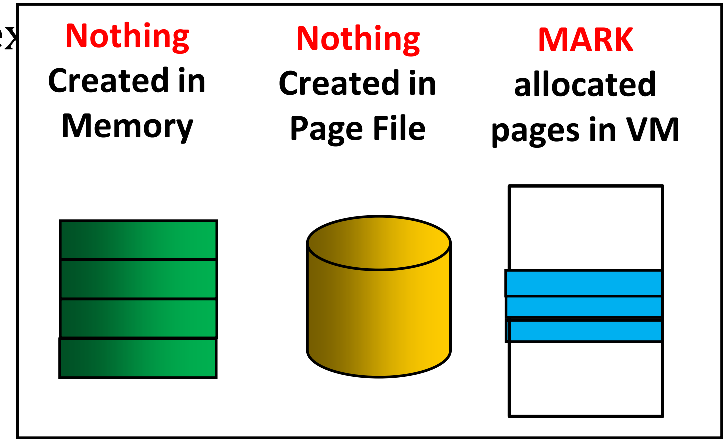
#2: allocate_user_mem()

```
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 (use **`PERM_UHPAGE`**)
2. **NOTE**: you can use **`create_page_table()`** to create non-ex



#3: free()

`void free(void* virtual_address)`

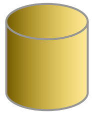
Description:

[USER SIDE] `lib/uheap.c`

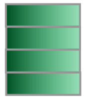
To access the environment data, use `myEnv` pointer

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

Refer to APPENDICES for:



Page File Helper Functions



Working Set Struct & Helper Func's

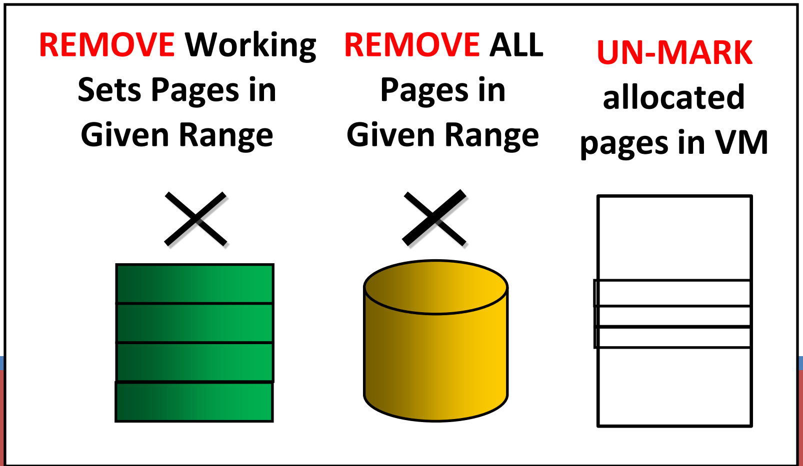
#4: free_user_mem()

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

Description:

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

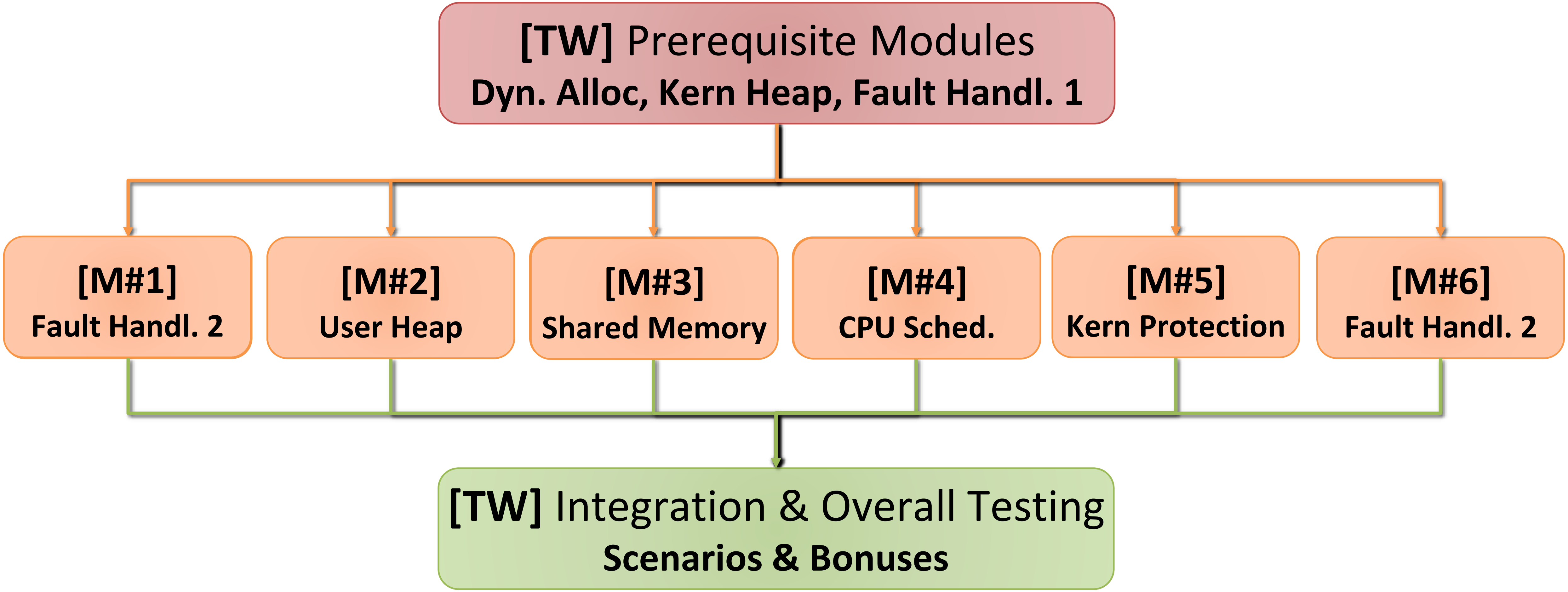
1. **Unmark** the given range to indicate it's **NOT reserved** for the page allocator of this environment
2. **Free ALL pages** of the given range from the **Page File**
3. **Free** ONLY pages that are resident in the **working set** from the memory



Shared Memory

PART III: INDIVIDUAL MODULE #3

Project Overview



Objective

Allow the two or more user processes to **share** any kind of data among them during the run-time

Shared Memory

REMEMBER

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

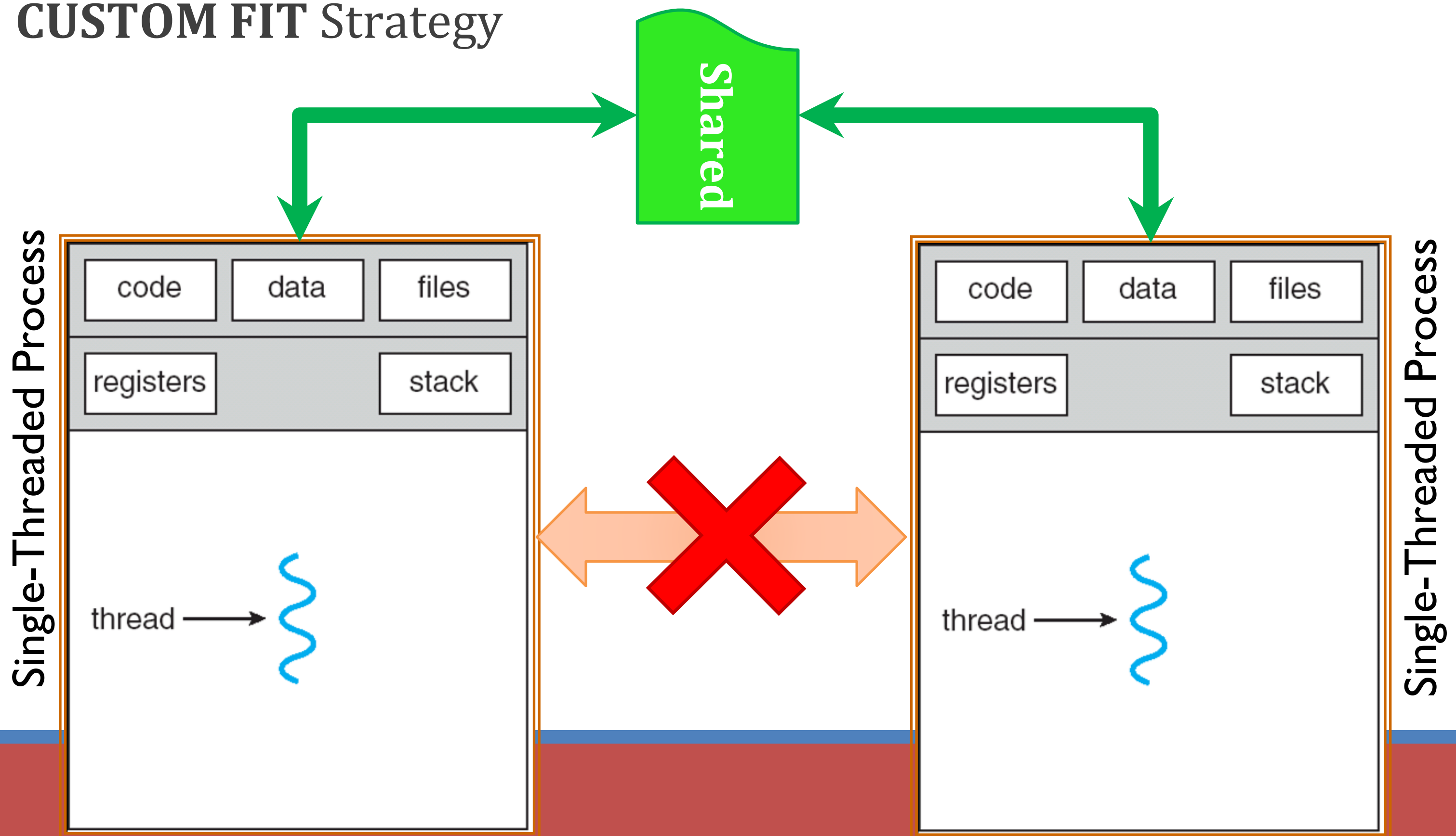
Shared Memory

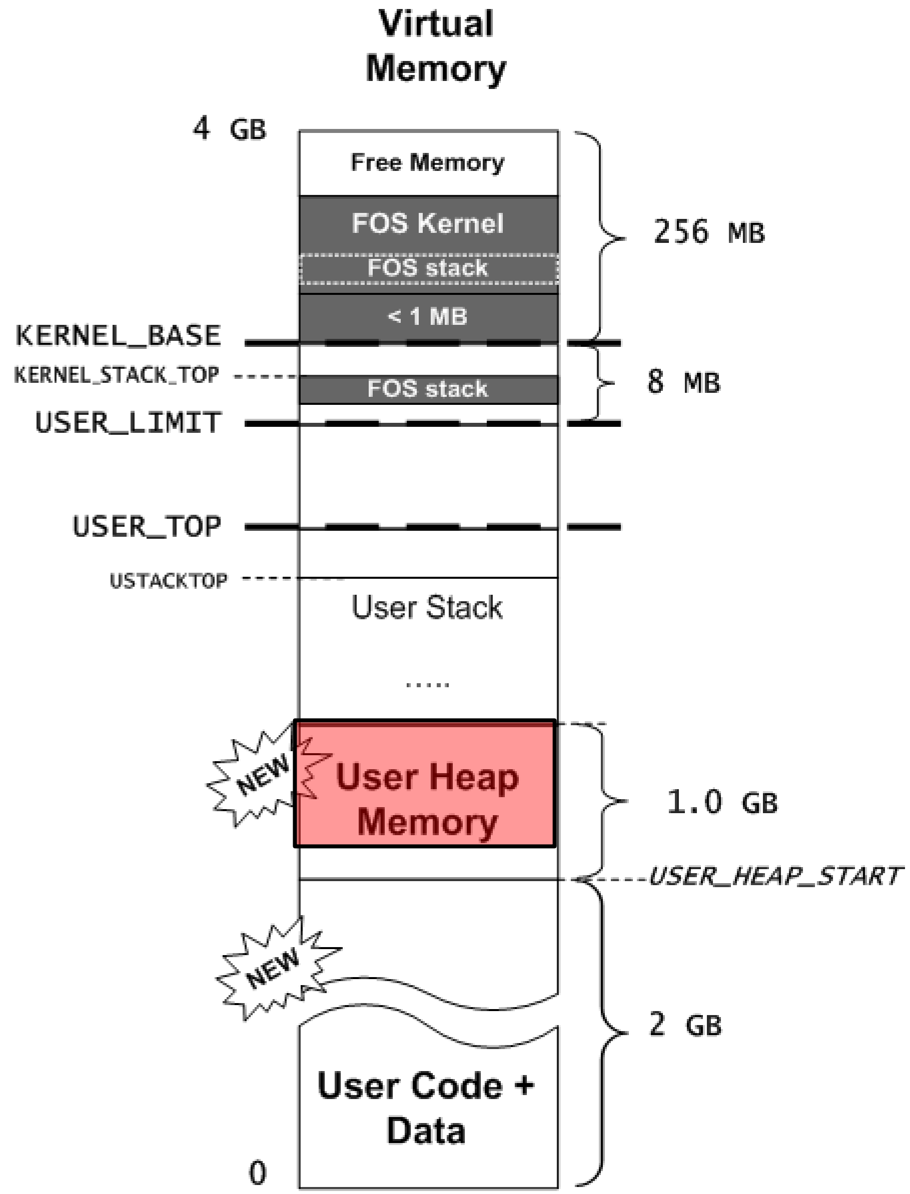
The main functions required to handle “**Shared Memory**” are:

#	Function	File
1	`smalloc (User side)`	**Functions definitions <u>TO DO</u> in: lib/uheap.c**
2	`sget (User side)`	
3	`alloc_share`	**Functions definitions <u>TO DO</u> in: kern/mem/shared_memory_manager.c**
4	`create_shared_object(Kernel side)`	
5	`get_shared_object(Kernel side)`	

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
  - **CUSTOM 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”);
```

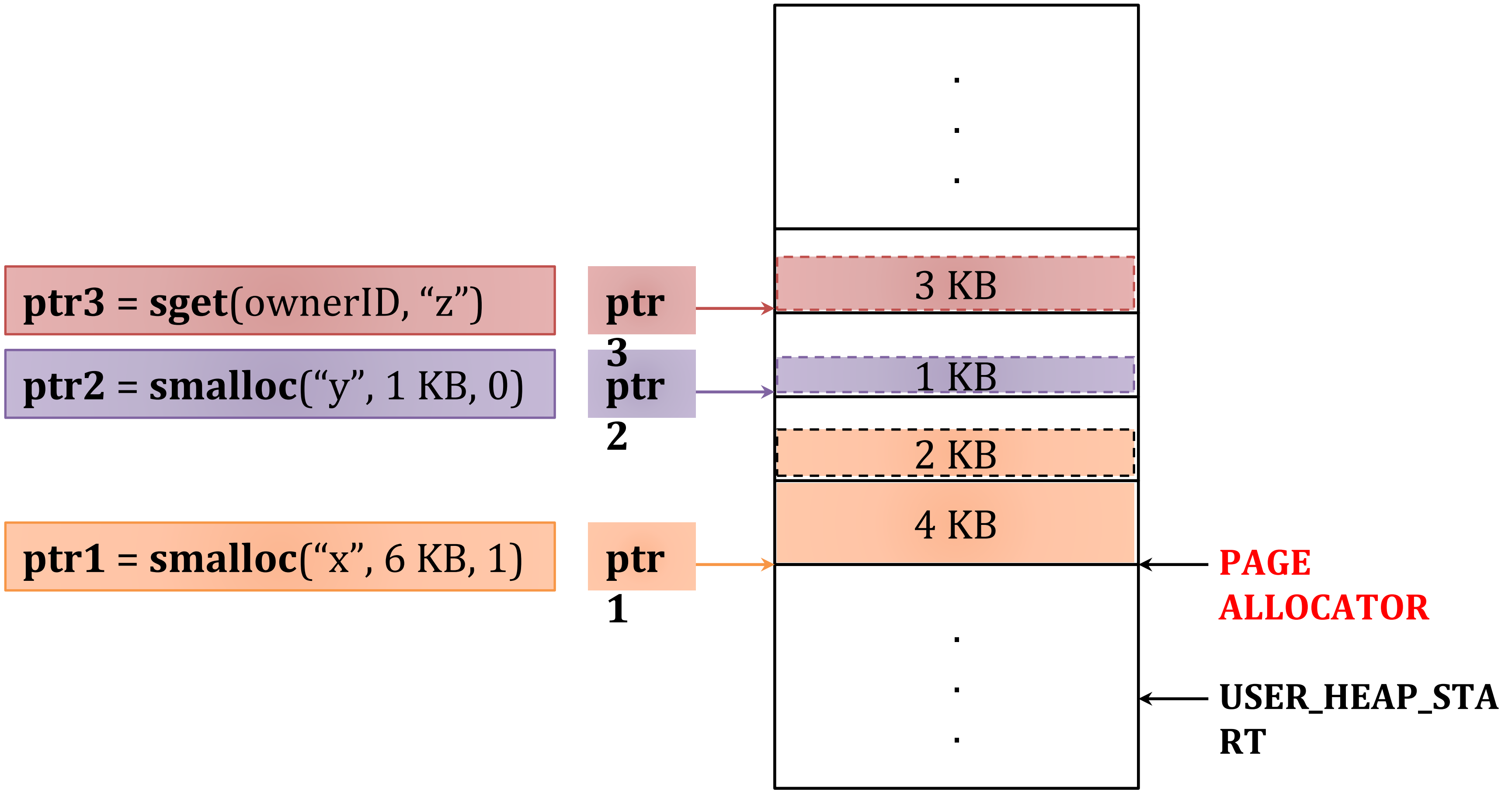
- Search for the shared object, named “mySharedInt” and belong to App1ID
- share it in app2, and return its virtual address in “ptr_sharedInt”

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

- Read its value (it should be 70)

Shared Memory: **Overview**

Create/Share pages on 4KB granularity



Shared Memory: **Overview**

CUSTOM FIT Strategy

1. Search for **EXACT fit**

2. if not found,

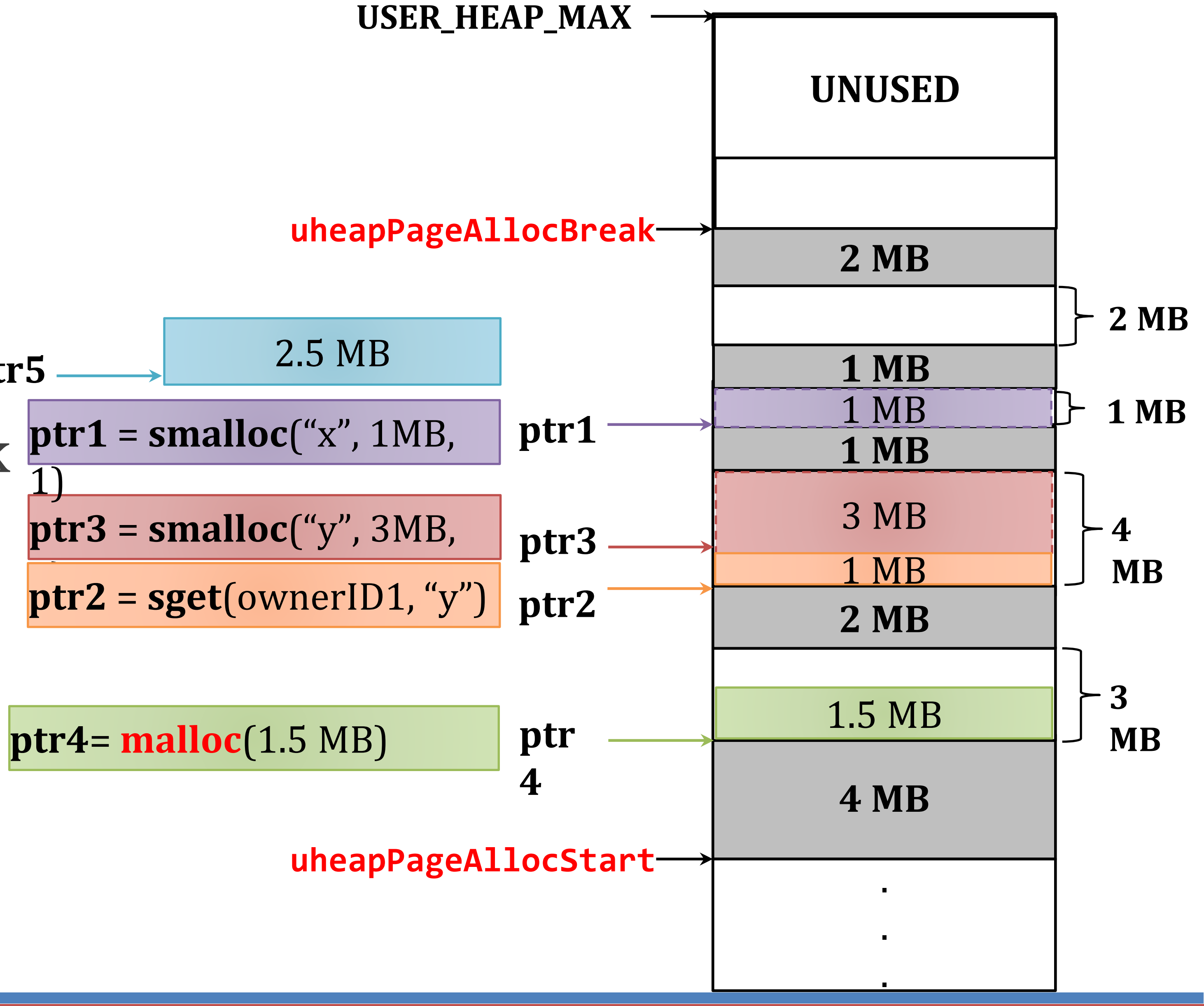
search for **WORST fit** till break

3. if not found,

extend **BREAK** if available

4. if not available,

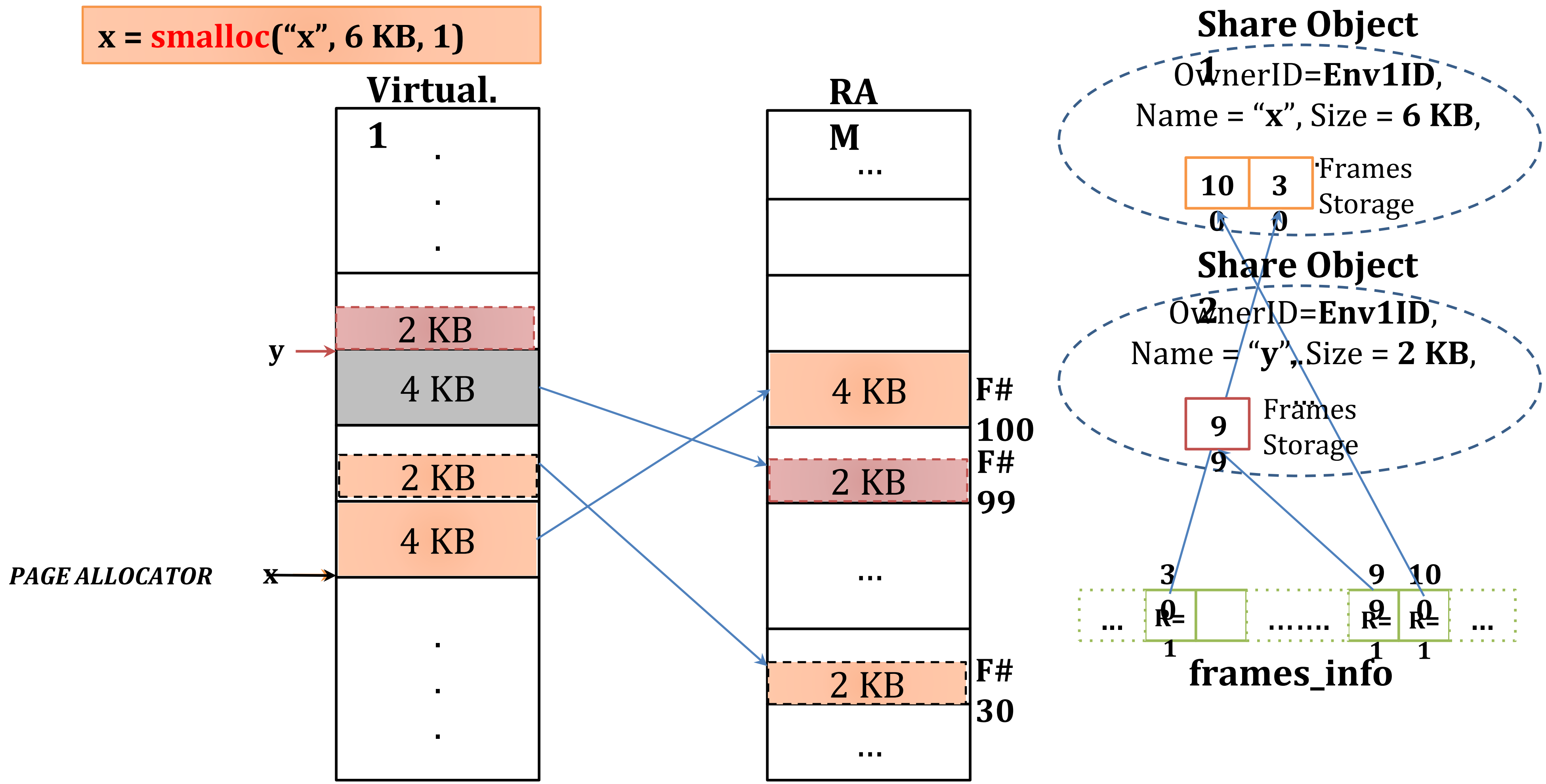
return **NULL**



Shared Memory: **Details**

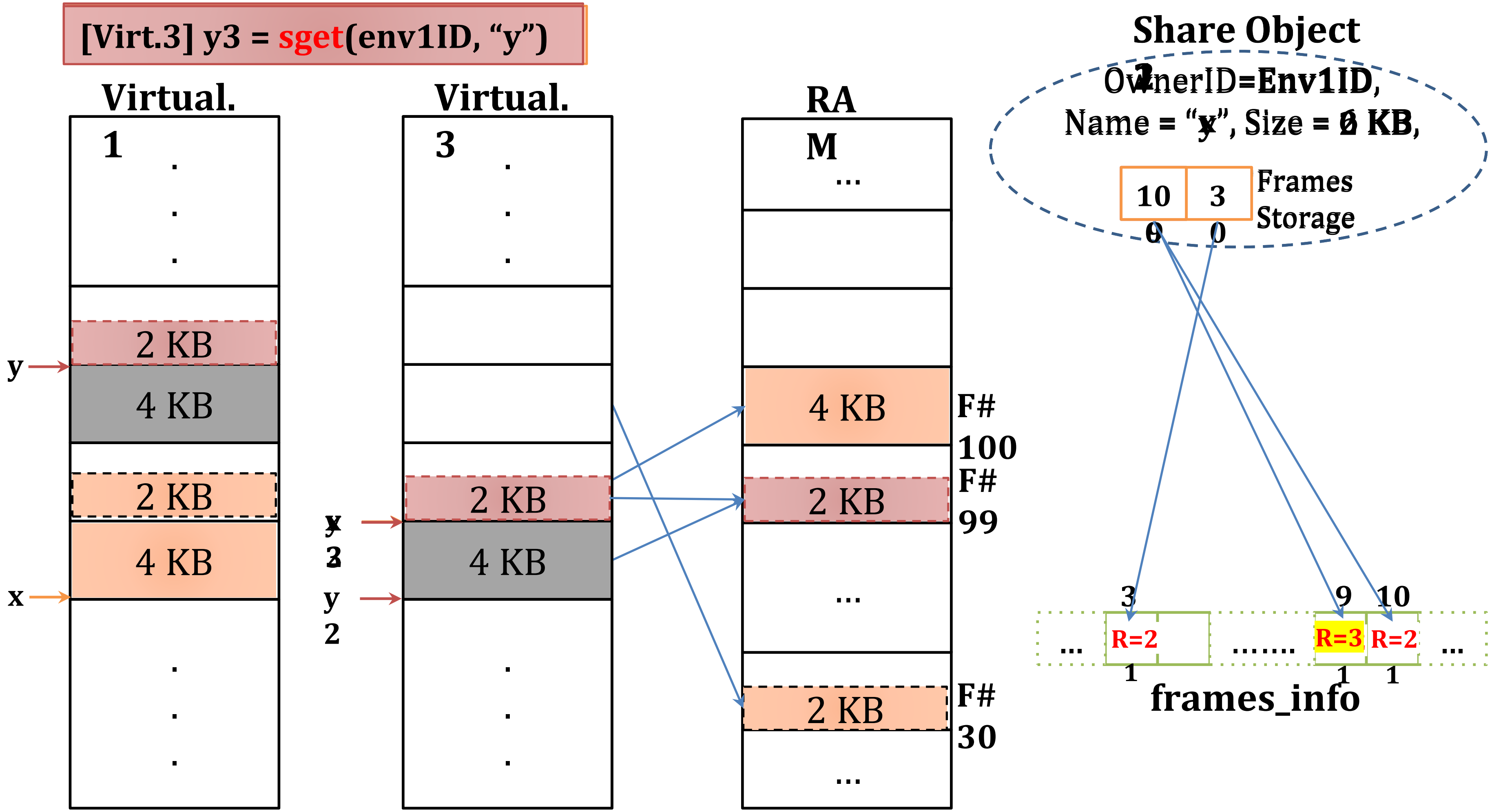
1. Space MUST be allocated in RAM
2. KEEP track of the allocated frames

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 ☺

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

(DONE)

- Search for shared object with the given “ownerID” & “name” in the “shares_list”
- If found: return pointer to the Share object, else: return NULL

```
int size_of_shared_object(int32 ownerID, char* shareName)
```

(DONE)

- Get the size of the shared object

kern/mem/shared_memory_manager.c

#1: Allocate & Initialize Share Object

```
struct Share* alloc_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-out its most significant bit
3. **Create** the "**framesStorage**": array of pointers to `struct FrameInfo` to save pointer(s) to the shared frame(s)
4. **Initialize** it by ZEROs
5. **Return**:
   1. If succeed: pointer to the created object for `struct Share`
   2. If failed: **UNDO** any allocation & **return** NULL

[USER SIDE] `lib/uheap.c`

#2: smalloc()

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

1. **Apply CUSTOM 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_create_shared_object`**(…) to invoke the Kernel for allocation of shared variable

RETURN:

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

#3: create_shared_object()

```
int create_shared_object(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

#4: sget()

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

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

RETURN:

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

#5: get_shared_object()

```c
int get_shared_object(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

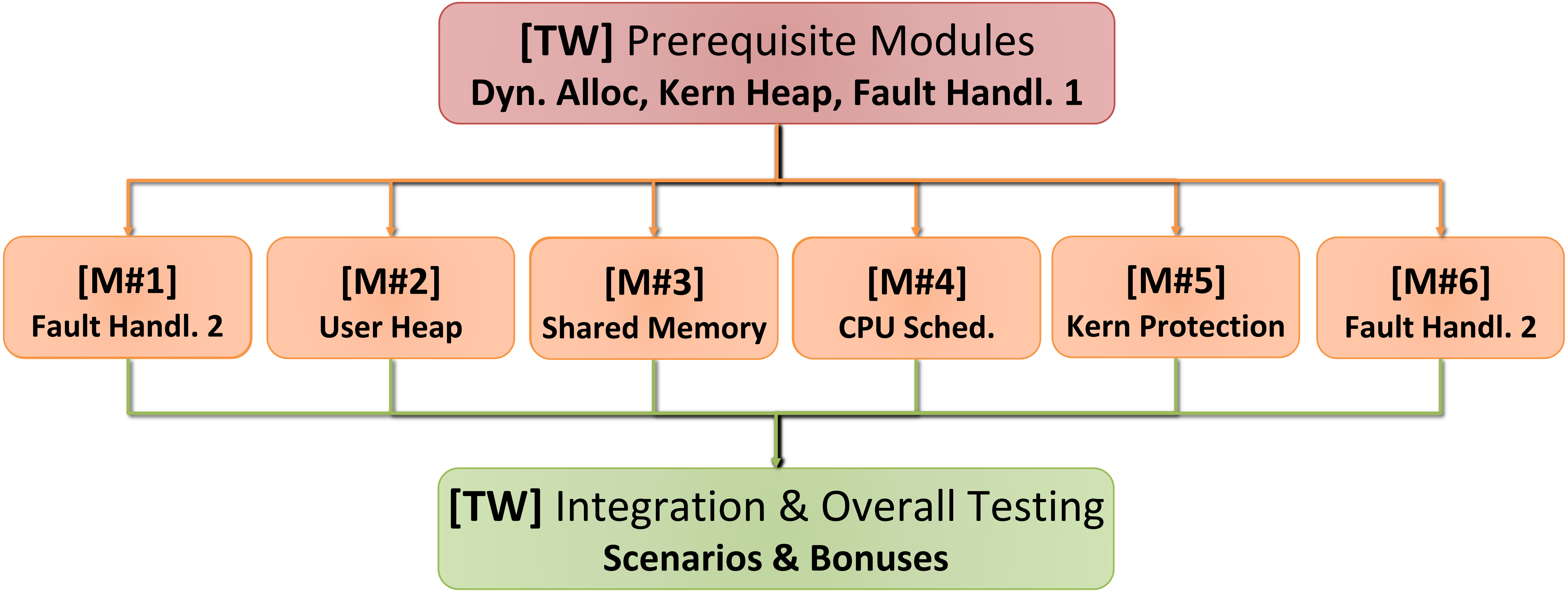
RETURN:

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**

CPU Scheduling

PART III: INDIVIDUAL MODULE #4

Project Overview



Objective

Support **Priority-based** **Round Robin** scheduling

among user processes & **avoid starvation**

Priority RR Scheduler

REMEMBER

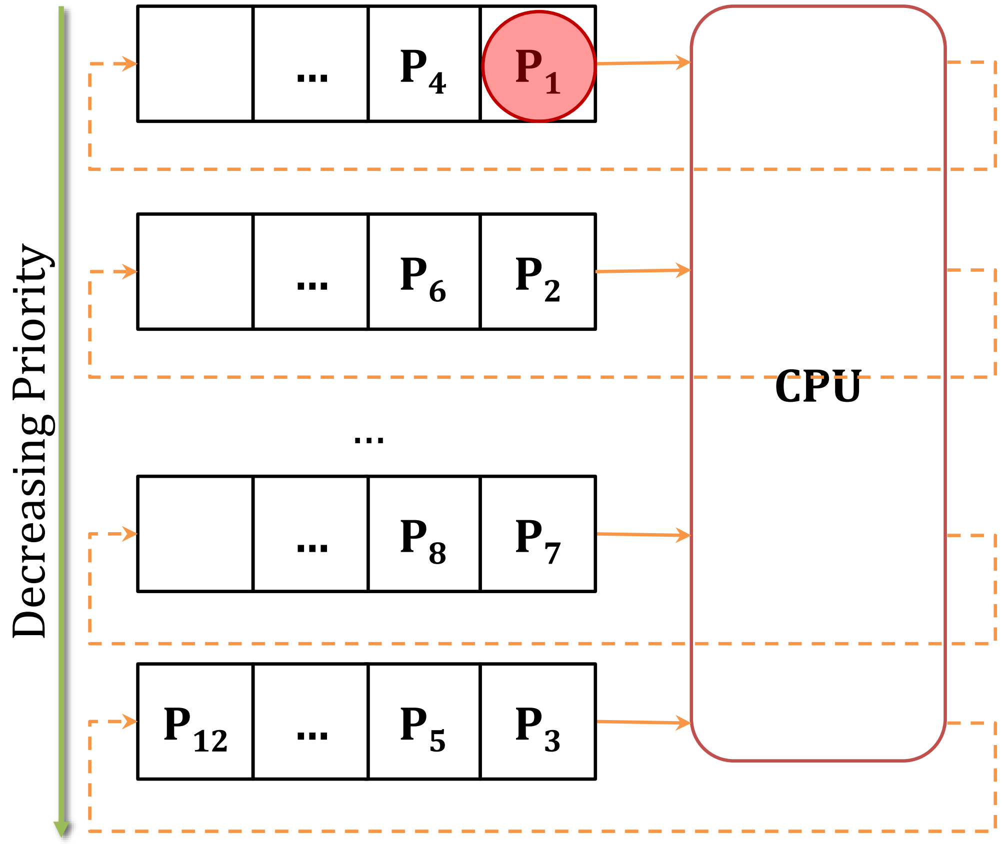
During your solution, any SHARED data need to be PROTECTED by critical section via LOCKS

Refer to APPENDICES for:

Scheduler Functions

Priority RR Scheduler: **Overview**

- **WHAT?**
  - multiple **ready queues** to represent each **level** of **priority**
  - Preemptive on clock
  - At any given time:
    - the scheduler chooses a process from the **highest-priority non-empty** queue.
    - If the highest-priority queue contains multiple processes, then they run in "**round robin**" order.

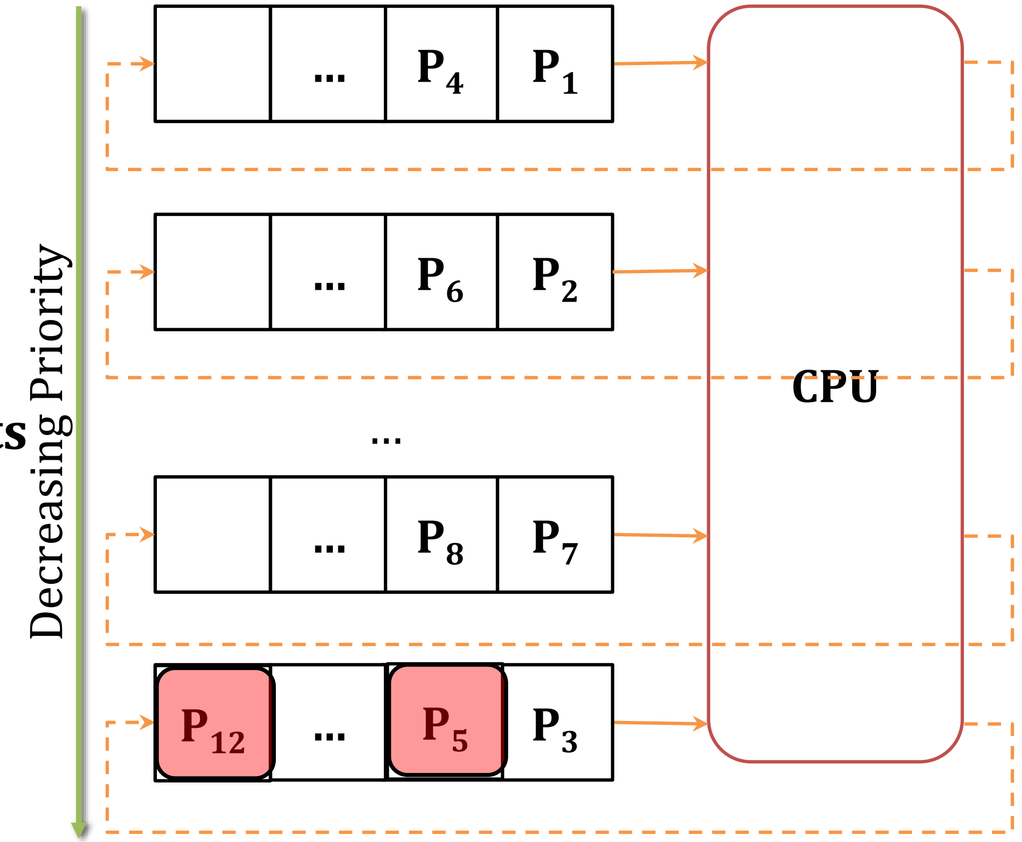


Refer to <u>APPENDICES</u> for:

Scheduler Functions

Priority RR Scheduler: **Overview**

- **PROBLEM**
  - Lower-priority may suffer **starvation** if there is a steady supply of high priority processes.
- **SOLUTION**
  - Allow a process to **change its priority** based on **its age** (IF #TICKS EXCEEDS CERTAIN THRESHOLD)



Given Data Structures & Functions

Refer to APPENDICES for:

Scheduler Functions

kern/cpu/sched.h

Queues

```
struct
{
    struct spinlock qlock;                    //SpinLock to protect all process queues
    struct Env_Queue env_new_queue;           // queue of all new envs
    struct Env_Queue env_exit_queue;          // queue of all exited envs
    struct Env_Queue *env_ready_queues;       // Ready queue(s) for the MLFQ or RR
}ProcessQueues;
```

CPU Scheduler

```
uint8 *quantums ;                             // Quantum(s) in ms
uint8 num_of_ready_queues ;                   // Number of ready queue(s)
```

Queues Function

kern/cpu/sched_helpers.c

```
void sched_insert_ready(struct Env* env);
```

Insert process in the correspond. ready queue (based on priority)

```
void sched_insert_exit(struct Env* env);
void sched_remove_exit(struct Env* env);
```

Sched Function

kern/cpu/sched_helpers.c

```
void sched_new_env(struct Env* e);
void sched_run_env(uint32 envId);
void sched_exit_env(uint32 envId);
void sched_kill_env(uint32 envId);
void sched_print_all();
void sched_run_all();
void sched_kill_all();
void sched_exit_all_ready_envs();
```

Given Commands

Refer to APPENDICES for:

Ready-Made Commands

1. **FOS>** run <prog_name> <page_WS_size> [<priority>]

2. **FOS>** load <prog_name> <page_WS_size> [<priority>]

3. **FOS>** setPri <envID> <priority>

4. **FOS>** setStarvThr <starvationThreshold>

5. **FOS>** runall

6. **FOS>** printall

7. **FOS>** sched?

#1: Set Process Priority & Thresh

Refer to APPENDICES for:

Scheduler Functions

```
void env_set_priority(int envID, int
                priority)
```

kern/cpu/sched_helpers.c

Description:

1. **Set** the priority of the given process by the given priority value
2. If it's in **READY state,** update its location in the ready queues

```
void sched_set_starv_thresh(uint32
                starvThresh)
```

Description:

- **Set** the starvation threshold by the given value

New System Call

```
void sys_env_set_priority(int32 envID, int
                priority)
```

Description:

- **Implement** and handle a new system call to set the **priority** of the user process from user level
- Should call the **env_set_priority(…)** from the kernel
- Should be named as **sys_env_set_priority(…)**
- **Refer** to MS#1 for steps

#2: Initialize Priority RR Scheduler

Refer to APPENDICES for:

Scheduler Functions

```
void sched_init_PRIRR(uint8 numOfPriorities, uint8 quantum, uint32 starvThresh)
```

kern/cpu/sched.c

Description:

- **Initialize** the Priority RR scheduler by the given number of priorities, CPU quantum (in millisecond) and starvation threshold
- Do other initializations (if any)
- Should use the following **global variables** for initialization (declared in **`kern/cpu/sched.h`**)

```
struct Env_Queue *env_ready_queues; // Ready queue(s)
uint8 *quantums ;                   // Quantum(s) in ms
uint8 num_of_ready_queues ;         // Number of ready queue(s)
```

Refer to APPENDICES for:

Scheduler Functions

#3: Schedule Next Process

```
struct Env* fos_scheduler_PRIRR()
```

Description:

- **If there's** a current **process on the CPU**, place it in the corresponding ready queue (do any required initializations)
- **Select** the next environment to be run on the CPU and return it
- **REMEMBER** to set the CPU quantum

Refer to **APPENDICES** for:

CPU Scheduler Functions

#4: Timer Tick Handler

```
void clock_interrupt_handler()
```

Description:

- This handler is automatically **called every "quantum"** period
- Should be used to **promote** any process that exceeds the **starvation threshold**
    - IF #TICKS IT EXCEEDS THE STARVATION THRESHOLD

Priority RR Scheduler: **Switching…**

- To switch the scheduler from the FOS prompt:
  - **FOS> schedPRIRR <#priorities> <quant> <starvThresh>**

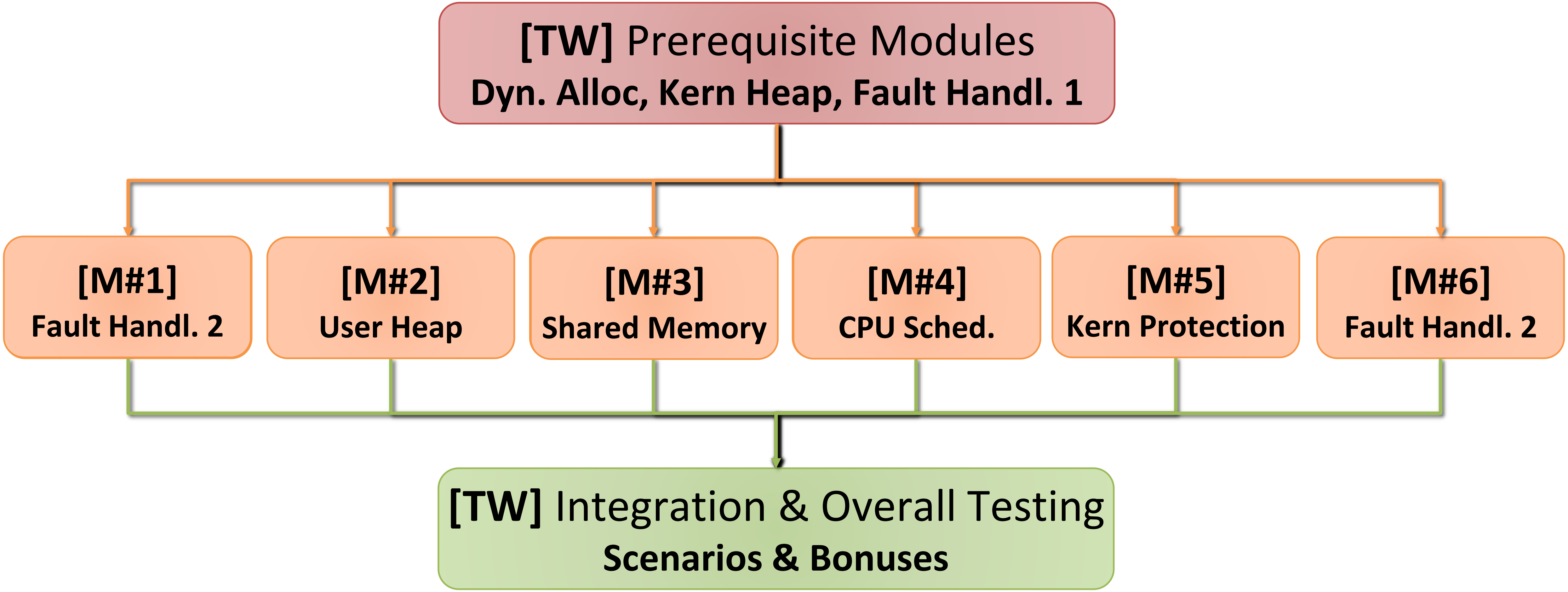
     switch the scheduler to Priority RR with the given #priorities, quantum and starvation threshold
  - **FOS> schedRR <quantum>**

     switch the scheduler to RR with the given quantum

Kernel Protection

PART III: INDIVIDUAL MODULE #5

Project Overview



Objective

Implement **two Kernel-based primitives** to support the **protection** of shared resources and/or the **blocking** on I/O

Kernel Protection

REMEMBER

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

Kernel Protection

The main functions required to implement “**SleepLocks**” & “**Semaphores**” are:

#	Function	File
1	`sleep()`	Declarations: **kern/conc/channel.h** **definitions <u>TO DO</u>: kern/conc/channel.c**
2	`wakeup_one()`	
3	`wakeup_all()`	
4	`acquire_sleeplock()`	Declarations: **kern/conc/sleeplock.h** **definitions <u>TO DO</u>: kern/conc/sleeplock.c**
5	`release_sleeplock()`	
6	`wait_ksemaphore()`	Declarations: **kern/conc/ksemaphore.h** **definitions <u>TO DO</u>: kern/conc/ksemaphore.c**
7	`signal_ksemaphore()`	

NOTE: MOST of them are small functions

1. SleepLocks: **Intro**

Locks provide two **atomic** operations:

- Lock.acquire() – **wait** until lock is **free**; then **mark** it as **busy**
  - After this returns, we say the calling thread ***holds*** the lock
- Lock.release() – **mark** lock as **free**
  - Should only be called by a thread that currently holds the lock
  - After this returns, the calling thread no longer holds the lock

Negatives of interrupt-based implementation:

- **Can't** give lock implementation to **users**
- **Doesn't** work well on **multiprocessor**

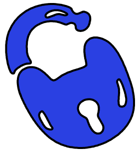
Negatives of SpinLocks:

Busy-waiting

Cache coherence

1. SleepLocks: **Implementation**

Idea: only busy-wait to atomically check lock value



```
SpinLock guard = FREE;
int mylock = FREE; //Interface: acquire(&mylock); release(&mylock);

acquire(int *thelock) {
        acquire_spinlock(&guard)
        while (*thelock == BUSY) {
                put thread on wait queue;
                go to sleep() & release guard
                // guard == BUSY on wakeup!
        }
        *thelock = BUSY;
        release_spinlock(&guard);
        }
}
```

```
release(int *thelock) {
        acquire_spinlock(&guard)
        if anyone on wait queue {
                wake-up ALL blocked
processes
        }
        *thelock = FREE;
        release_spinlock(&guard);
}
```

Note: unlike previous solution, the critical section is **very short**

Note: WHY there's a `while` (not `if`) in the `acquire`?

Note: `sleep` must be sure to **reset the guard variable. WHY** can't we do it just before or after the sleep?

1. SleepLocks: **Implementation**

What about **release guard** when going to sleep?

```
SpinLock guard = FREE;
int mylock = FREE;

acquire(int *thelock) {
        acquire_spinlock(&guard)
        while (*thelock == BUSY) {
                put thread on wait queue;
                    & release guard
                go to sleep()
                // guard == BUSY on wakeup!
        }
        *thelock = BUSY;
        release_spinlock(&guard);
        }
}
```

Release Position
Release Position
Release Position

Before Putting thread on the wait queue?

- Release can check the queue and not wake up thread **missing wake-up**

After putting the thread on the wait queue

- Release puts the thread on the ready queue, but the thread still thinks it needs to go to sleep
- **Miss wakeup** and still holds lock (**deadlock**!)

Want to put it after `sleep()`. But – **how?**

- In Sleep: **Protect** the process queue(s) then **release** the guard. This ensures **no missing wake-up** even if release is called.

1. SleepLocks: **Implementation**

Positives

- Machine can receive **interrupts**
- **User** code can use this lock
- Works on a **multiprocessor**
- No **Busy-Waiting**

Negatives

- Need **system call** in sleep() & wakeup()
  - Min System call ~ 25x cost of function call

Usually used for
long-time critical section

EX: INITIALIZING 2D MATRIX 2000x2000

1. **SYSTEM CALL RANDOM FUNCTION ⍰ 190 secs**
2. **USER-SIDE RANDOM FUNCTION ⍰ 5 secs**

SPEEDUP FACTOR ≈ 38x

1. SleepLocks: Given Data Structures

1. Complete implementation of **SpinLock** (kern/conc/spinlock.h & .c)

2. **SpinLock** to protect ANY process queue (kern/cpu/sched.h)

ProcessQueues.qlock

1. Struct declaration of the **SleepLock** (kern/conc/sleeplock.h)

```
struct sleeplock
{
    bool locked;              // Is the lock held?
    struct kspinlock lk;      // spinlock protecting this sleep lock
    struct Channel chan;      // channel to hold all blocked processes on this lock
    // For debugging:
    char name[NAMELEN];       // Name of lock.
    int pid;                  // Process holding lock
};
```

1. Struct declaration of the **Channel** (kern/conc/channel.h)

```
struct Channel
{
    struct Env_Queue queue; //queue of blocked processes waiting on this channel
    char name[NAMELEN];     //channel name
};
```

5. Process Status (inc/environment_definitions.h)

```
// Values of env_status in
#define ENV_FREE        0
#define ENV_READY       1
#define ENV_RUNNING     2
#define ENV_BLOCKED     3
#define ENV_NEW         4
#define ENV_EXIT        5
#define ENV_UNKNOWN     6
```

1. SleepLocks: Given Functions

1. Initialize the **SleepLock** (kern/conc/sleeplock.c)

```
void init_sleeplock(struct sleeplock *lk, char *name)
{
    init_channel(&(lk->chan), "sleep lock channel");
    init_spinlock(&(lk->lk), "lock of sleep lock");
    strcpy(lk->name, name);
    lk->locked = 0;
    lk->pid = 0;
}
```

1. Check whether the lock is held or not? (kern/conc/sleeplock.c)

```
int holding_sleeplock(struct sleeplock *lk)
{
    int r;
    acquire_spinlock(&(lk->lk));
    r = lk->locked && (lk->pid == get_cpu_proc()->env_id);
    release_spinlock(&(lk->lk));
    return r;
}
```

1. SleepLocks: **Given Functions**

3. Get the current running process (kern/proc/user_environment..c)

```
struct Env* get_cpu_proc();
```

3. Insert a process into the ready queue (kern/cpu/sched_helpers.c)

```
void sched_insert_ready(struct Env* p);
```

3. Queues functions: (kern/cpu/sched_helpers.c)

```
int queue_size(struct Env_Queue* queue);
void enqueue(struct Env_Queue* queue, struct Env* env);
struct Env* dequeue(struct Env_Queue* queue);
```

3. Invoke the scheduler to context switch into the next ready queue (if any) (kern/proc/user_environment..c)

```
void sched();
```

1. SleepLocks: **Required Functions**

#1: Sleep on Channel

```
void sleep(struct Channel *chan, struct spinlock* lk);
```

Description:

- Should **block** the current running process on the given `chan`and **schedule** a next ready one
- It should **release** the given `lk` before being blocked so that other process(es) can use it
- It should **reacquire** the given `lk` again when awakened.

GENERAL NOTE: make sure to protect any process queue using the suitable lock

1. SleepLocks: Required Functions

#2: Wake-up ONE in Channel

```
void wakeup_one(struct Channel *chan)
```

Description:

- Should **wake-up ONE blocked** process in the given **chan** and change it to **ready**

GENERAL NOTE: make sure to protect any process queue using the suitable lock

1. SleepLocks: **Required Functions**

#3: Wake-up ALL in Channel

```
void wakeup_all(struct Channel *chan)
```

Description:

- Should **wake-up ALL blocked** process(es) in the given `chan` and change them to **ready**

GENERAL NOTE: make sure to protect any process queue using the suitable lock

1. SleepLocks: **Required Functions**

#4: Acquire Sleep Lock

```
void acquire_sleeplock(struct sleeplock *lk)
```

Description:

- Should **acquire** the given sleep lock **`lk`**
- If successfully acquired, continue
- If failed, block the process on the corresponding channel
- Refer to the previously explained pseudocode

GENERAL NOTE: make sure to protect any process queue using the suitable lock

1. SleepLocks: **Required Functions**

#5: Release Sleep Lock

```
void release_sleeplock(struct sleeplock *lk)
```

Description:

- Should **release** the given sleep lock **`lk`** by **waken-up ALL blocked** processes on it
- Refer to the previously explained pseudocode

GENERAL NOTE: make sure to protect any process queue using the suitable lock

2. Semaphores: **Intro**

Operations:

1. **Initialize** by **non-negative** value
2. **semWait:**
    1. decrements the value
    2. If value < 0 ⍰ process is blocked
    3. Else ⍰ process continues execution
3. **semSignal:**
    1. increments the value.
    2. If value ≤ 0 ⍰ unblock a process (make it Ready)

3 Operations are **atomic**

2. Semaphores: **Intro**

```
struct semaphore {
    int count; lock = 0
    queueType queue;};
```

```
           void semWait(semaphore s)
           {
acquire()  [  int keyw = 1;
              do xchg(&keyw, &s.lock) while (keyw != 0);
              s.count--;
              if (s.count < 0) {
                 /* place this process in s.queue */;
                 /* block this process (must also set s.lock = 0) */; WHERE?
              }
release()  [  s.lock = 0;        }
           void semSignal(semaphore s)
           {
acquire()  [  int keys = 1;
              do xchg(&keys, &s.lock) while (keys != 0);
              s.count++;
              if (s.count ≤ 0) {
                 /* remove a process P from s.queue */;
                 /* place process P on ready list */;}
release()  [  s.lock = 0;        }
```

2. Semaphores: **Intro**

Usage:

1. Critical Section

```
//Semaphore for critical section

Semaphore S = 1 ; //only 1 process can enter critical section
```

Function1()	Function2()

<pre>Function1()
{
    ...
 	
    S.Wait()
 	
    <critical section>
 	
    S.Signal()
 	
    ...
}</pre>
	<pre>Function2()
	{
	    S.Wait()
	
	    <critical section>
	
	    S.Signal()
	
	    ...
	}</pre>

2. Semaphores: **Intro**

Usage:

1. Synchronization

```
//Semaphore for dependency (Function1 depends on Function2)

Semaphore D1 = 0 ; //block first until released

Semaphore D2 = 20 ; //start first until blocked
```

Function1()	Function2()
{	{
...	...
D1.Wait()	**Required Code**
Dependent code	D1.Signal()
...	}
}	

2. Semaphores: **Intro**

Pros:

1. **No busy-waiting**
2. Applicable to **any number of processes**
3. Applicable on **Uni or Multi processors**
4. Support **multiple critical sections**
5. **No starvation**
   - selection of a waiting process is **FIFO**
6. **No deadlock** in **Priority Inversion** (if code is correctly written!)
   - when a **low-priority** process interrupted inside CS,
   - High-priority can't enter CS, will be BLOCKED state
   - Low-priority can continue

2. Semaphores: **Intro**

Cons:

1. Semaphores are **dual** purpose, slight change in order of wait's deadlock!!
2. **Deadlock** in **Priority Inversion** of its **lock**
    - when a **low-priority** process acquires the lock then interrupted,
    - High-priority executes busy-waiting on the lock
    - Low-priority can't resume! Deadlock!

Solutions:

1. priority **promotion**
2. Priority **donation**

2. Semaphores: **Given** Data Structures

1. Struct declaration of the **`ksemaphore`** (kern/conc/ksemaphore.h)

```
struct ksemaphore
{
    int count;                  // Semaphore value
    struct kspinlock lk;        // spinlock protecting this count
    struct Channel chan;        // channel to hold all blocked processes on this sema
    // For debugging:
    char name[NAMELEN];         // Name of semaphore.
};
```

2. Semaphores: **Given** **Functions**

1. Initialize the **`ksemaphore`** (kern/conc/ksemaphore.c)

```c
void init_ksemaphore(struct ksemaphore *ksem, int value, char *name)
{
    init_channel(&(ksem->chan), "ksemaphore channel");
    init_kspinlock(&(ksem->lk), "lock of ksemaphore");
    strcpy(ksem->name, name);
    ksem->count = value;
}
```

2. Semaphores: Required Functions
#6: Wait

kern/conc/ksemaphore.c

```
void wait_ksemaphore(struct ksemaphore sem)
```

- Implement the logic of the "**wait**" function
- Refer to the previous pseudocode for details

MAKE USE OF YOUR CHANNEL IMPLEMENTATION

2. Semaphores: **Required Functions**

#7: Signal

kern/conc/ksemaphore.c

```
void signal_ksemaphore(struct ksemaphore sem)
```

- Implement the logic of the “**signal**” function
- Refer to the previous pseudocode for details

MAKE USE OF YOUR CHANNEL IMPLEMENTATION

FINALLY, LET'S RUN THE **ENTIRE FOS**...

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