Operating Systems’22

Project Description

Agenda

- Logistics
- What’s New?
- [**Kernel**] Project Features
  1. Kernel Heap
  2. Page Fault Handler
  3. CPU Scheduling
- [**User**] Project Features
  1. User Heap
- Bonuses
- CHALLENGES

Logistics

- **Group Members:** 3-5
- **Group Registration:**

  due to **WED 20 APR 23:59**

  - **Group of 6 members is asked to implement one of the bonus tasks as MANDATORY**

  – Register by **student ID**
- **Startup Code**:
  – FOS_PROJECT_2022_template.zip
  – Follow these steps to import the project folder into the eclipse

Logistics

ADVICE#1: WORK AS A TEAM

- Project Functions:

1.	Kernel Heap	→	5 functions
2.	Page Fault Handler	→	2 cases
3.	Create Page table	→	1 function
4.	Scheduler	→	2 functions
5.	User Heap	→	4 functions
–	TOTAL	=	**13 tasks**

- Average # Tasks / Member = 13 / 4 members

≈ **3.25 Tasks** on **4 Wee**

≈ **1 ASSIGNMENT**

Logistics

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

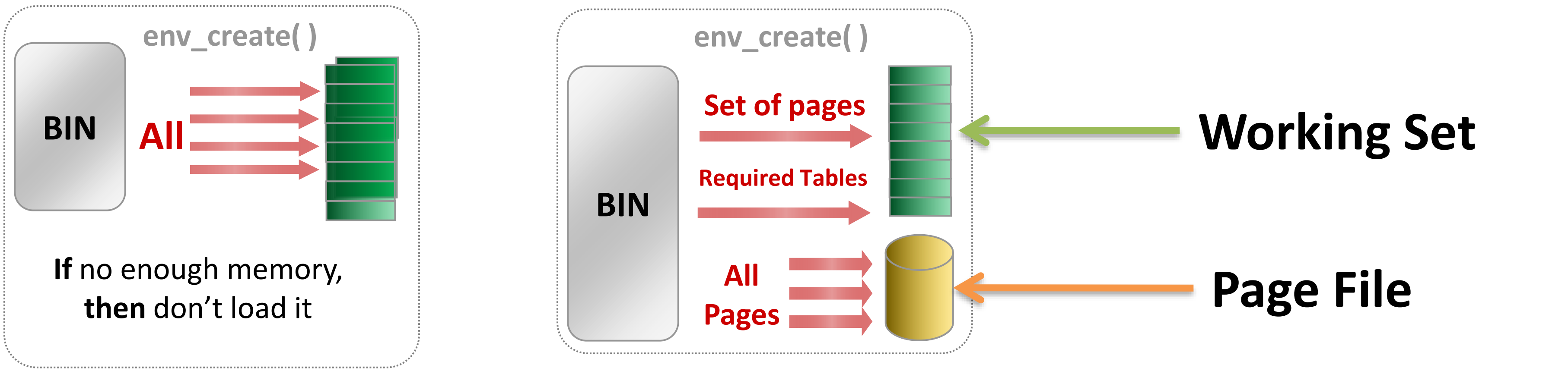
- **GUIDES:**
  - **70%** are following steps
  - **30%** invent your own solution
  - **ADVICE#2: MUST** read the **documentation** for
    - Detailed steps
    - Helper functions (*appendices*)

What’s New?

OLD

NEW

NEW Concepts



env_create( )

BIN

All

If no enough memory,
then don’t load it

env_create( )

BIN

Set of pages

Required Tables

All Pages

Working Set

Page File

Refer to the

Project Documentation

Project Features

[KERNEL]

1. **Kernel Heap:** dynamic allocation and free
   - **NEXT FIT** strategy
2. **Load and run** multiple user programs **(*mostly DONE*)**
3. **Page fault handler**
   - **MODIFIED CLOCK** replacement algorithm
4. **CPU Scheduling:** multi-level feedback queue

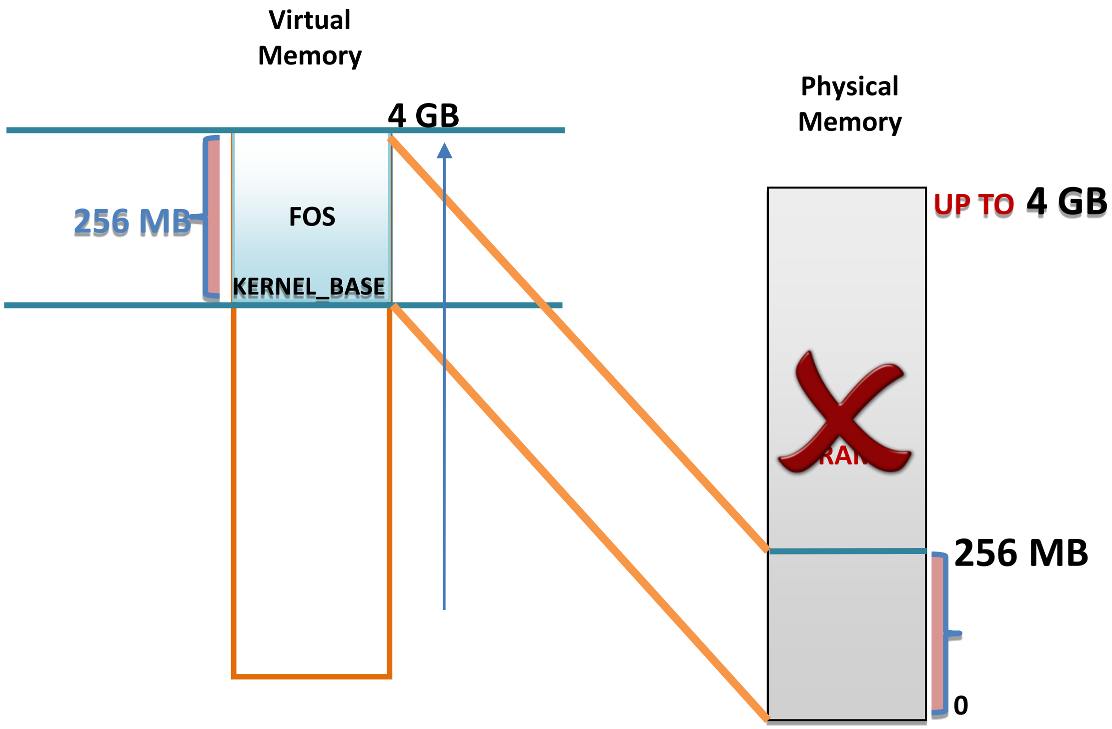
[USER]

1. **User Heap:** dynamic allocation and free
   - **NEXT FIT** strategy

Kernel Heap

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

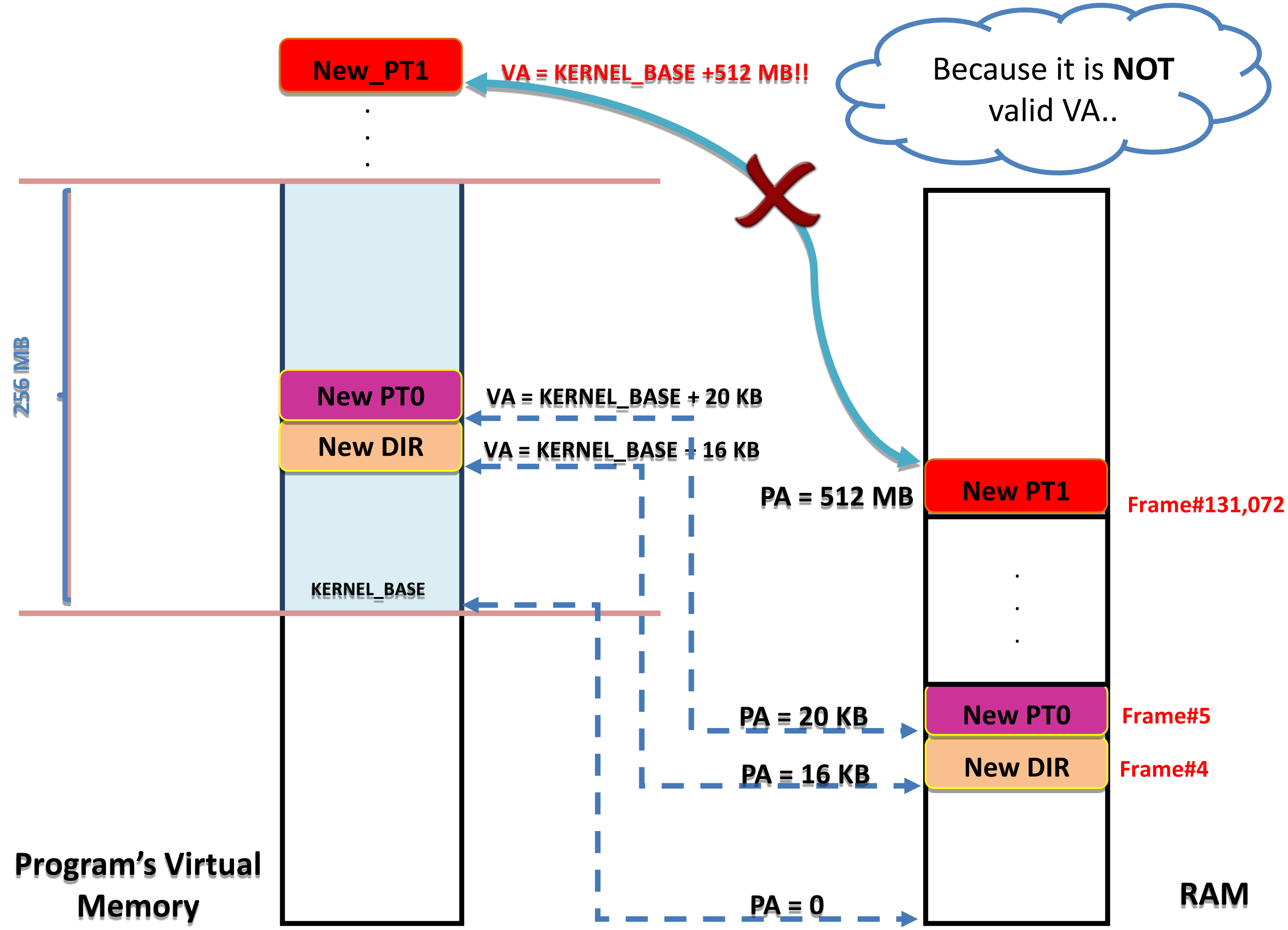
Kernel can’t directly access beyond 256 MB RAM



Kernel Heap

- Example:

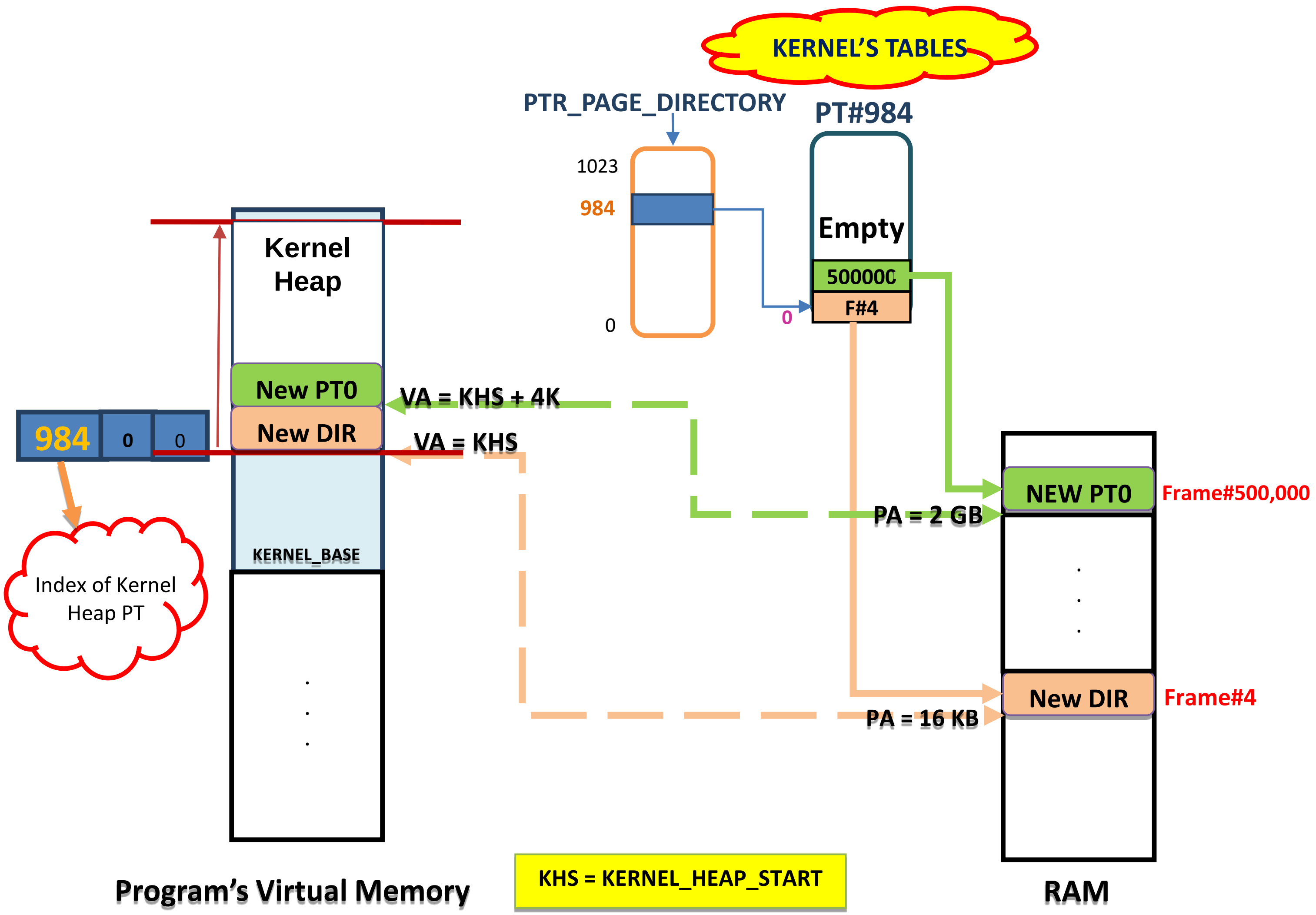
Kernel can’t directly access beyond 256 MB RAM



Kernel Heap

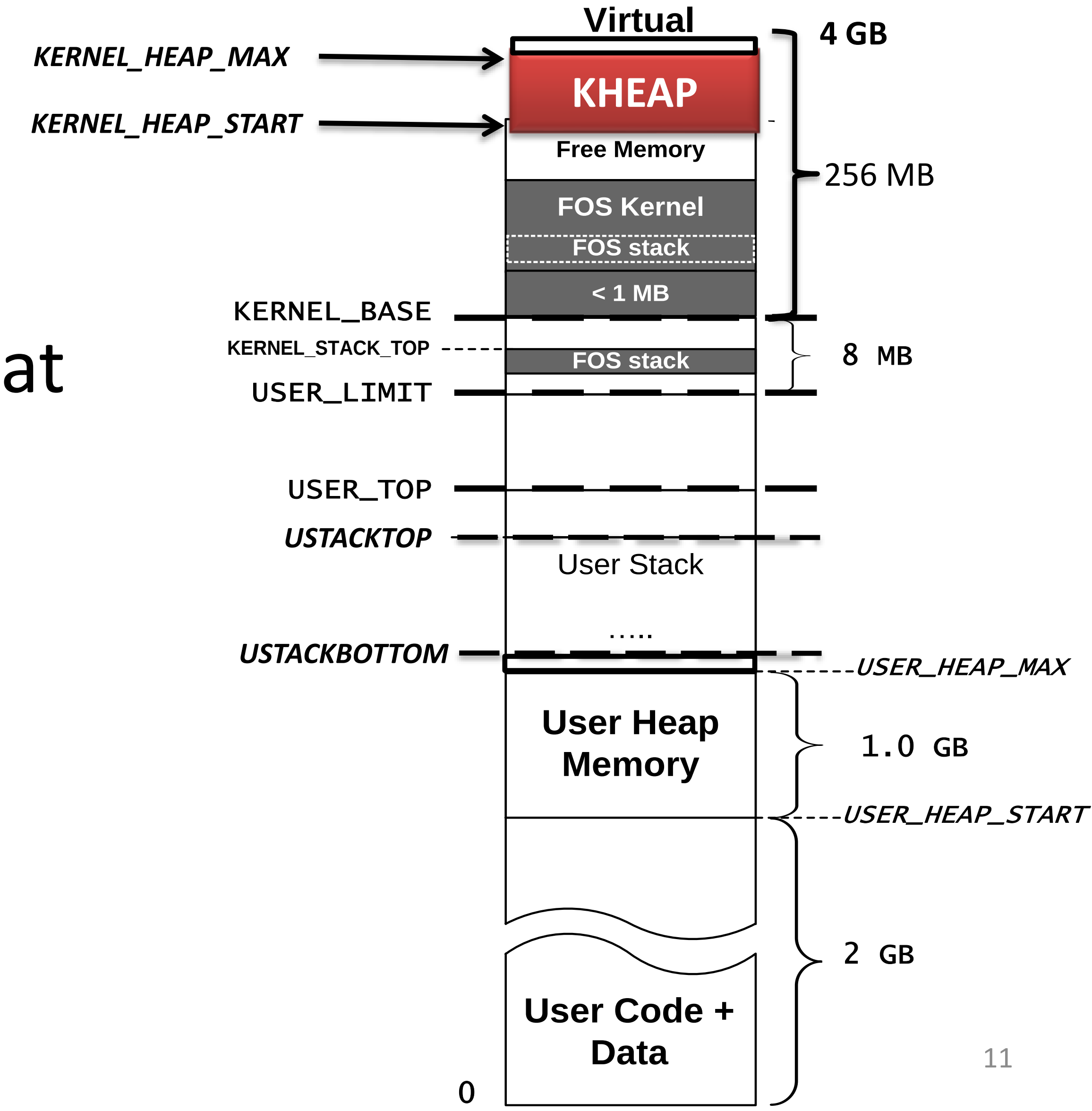
- Solution:

Kernel Heap for dynamic allocations (No 1-1 map)



Kernel Heap

- Kernel Heap lies at the end of the virtual space



Kernel Heap

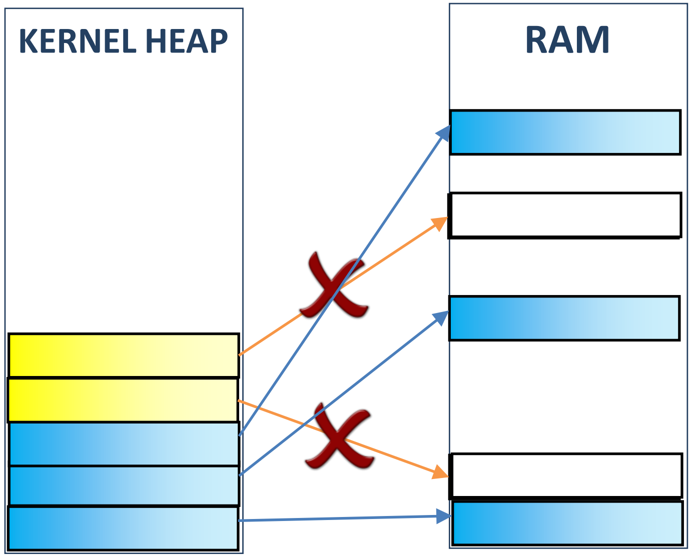
1. **Kmalloc():** dynamically allocate space
2. **Kfree():** delete a previously allocated space

kmalloc()

ALL Pages Created In Memory

Kfree()

Remove Pages BUT NOT Tables



Kernel Heap

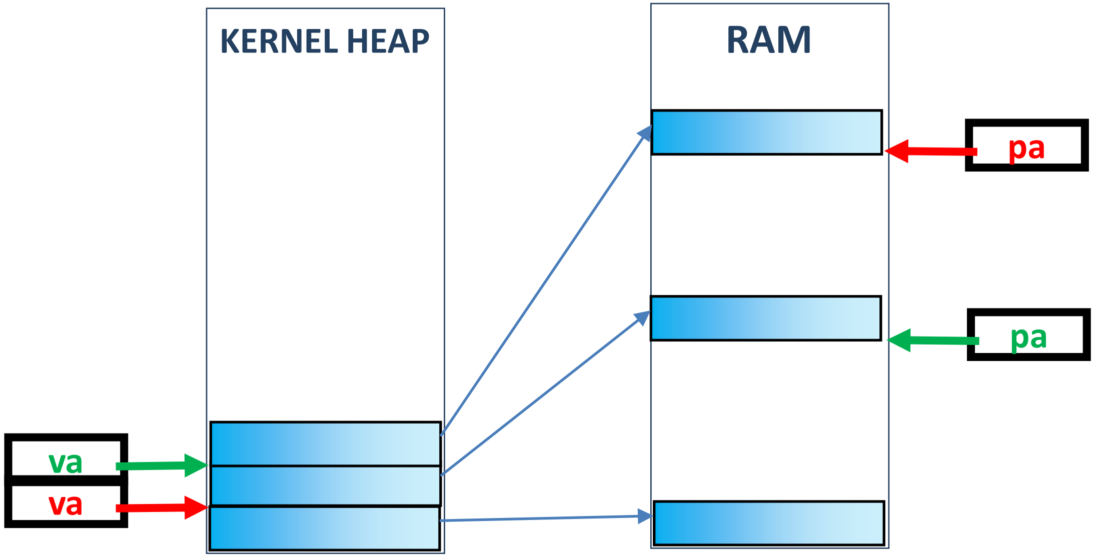
3. **Kheap_physical_address():** find physical address of the given kernel virtual address
4. **Kheap_virtual_address():** find kernel virtual address of the given physical one

kheap_physical_address()

Get pa of the given va

kheap_virtual_address()

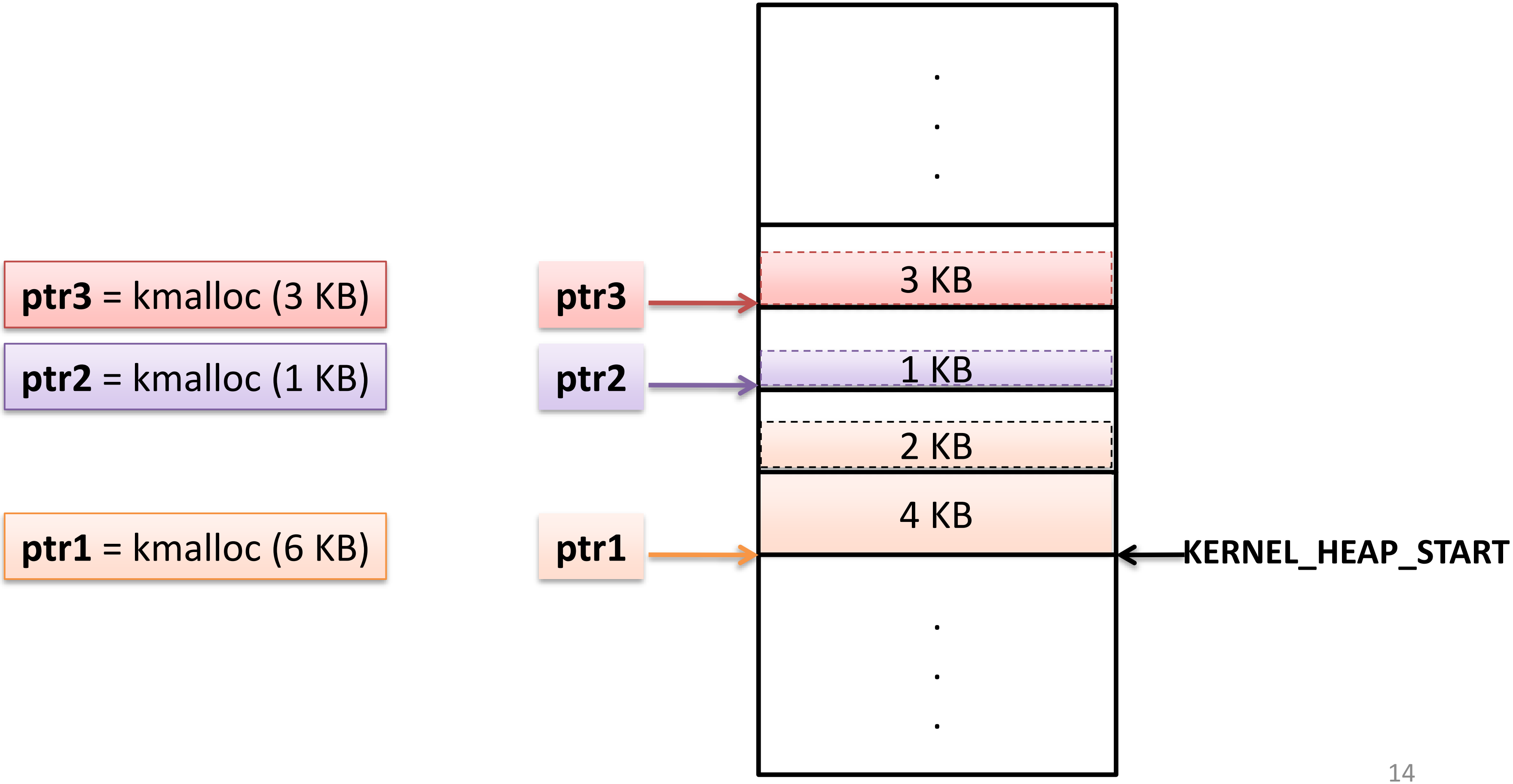
Get va of the given pa



Kernel Heap

[kmalloc() / kfree()]

- **Allocate pages on 4KB granularity**



Dynamic allocation/Deallocation [malloc() / free()]

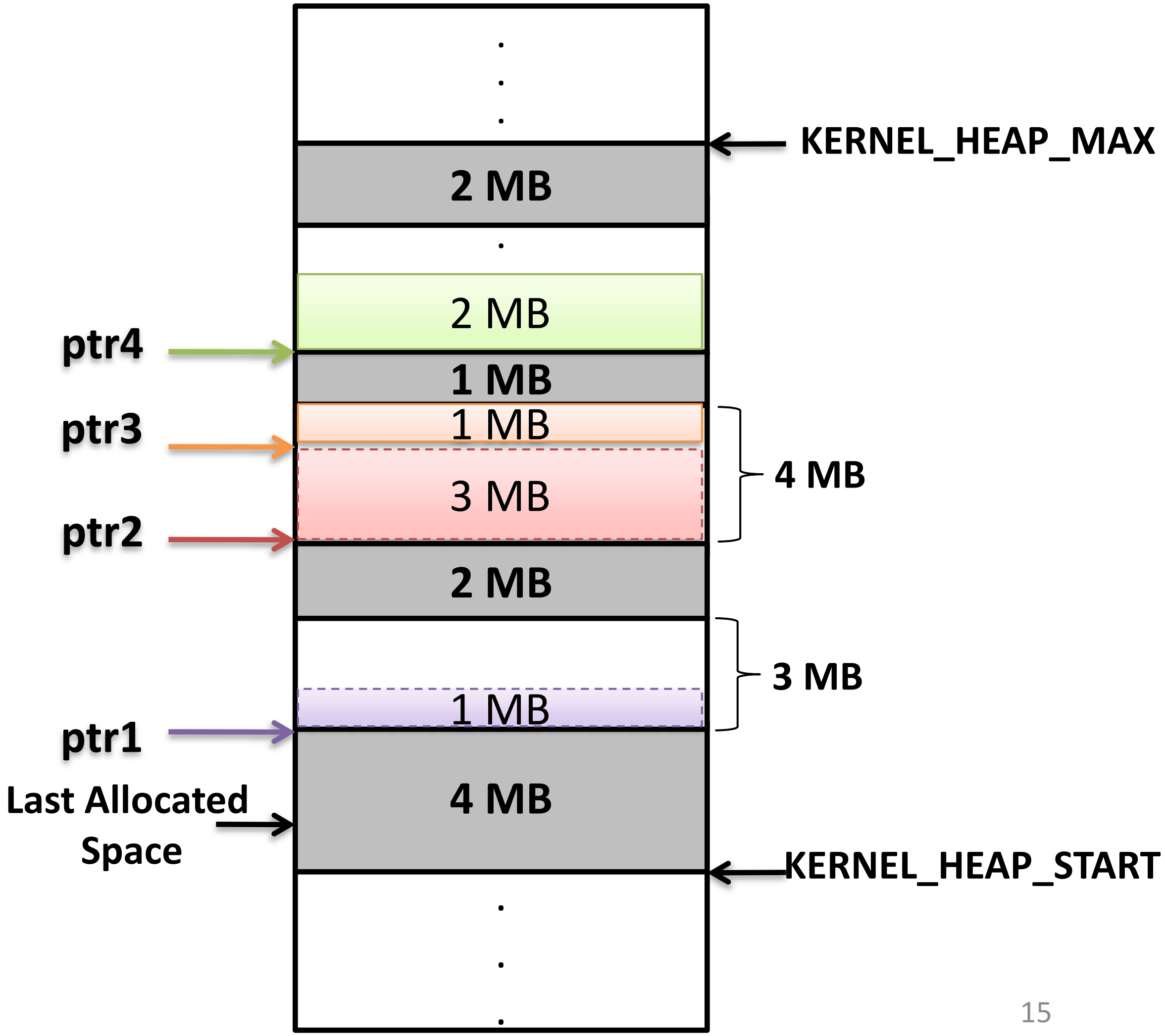
- **NEXT FIT Strategy**

ptr4 = kmalloc (2 MB)

ptr3 = kmalloc (1 MB)

ptr2 = kmalloc (3 MB)

ptr1 = kmalloc (1 MB)



Project Features

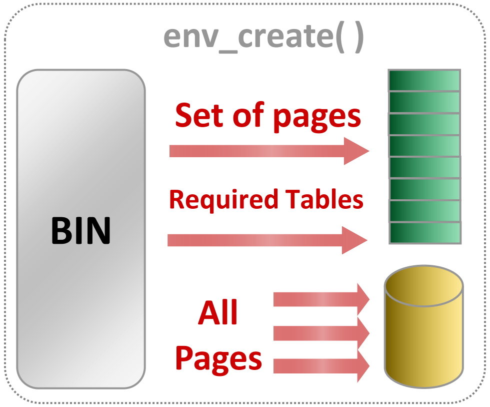
[KERNEL]

1. **Kernel Heap:** dynamic allocation and free
   - **BEST FIT** strategy
2. **Load and run** multiple user programs (***mostly DONE***)
3. **Page fault handler**
   - **MODIFIED CLOCK** replacement algorithm
4. **CPU Scheduling:** multi-level feedback queue

[USER]

1. **User Heap:** dynamic allocation and free
   - **BEST FIT** strategy

Loading Program (env_create)



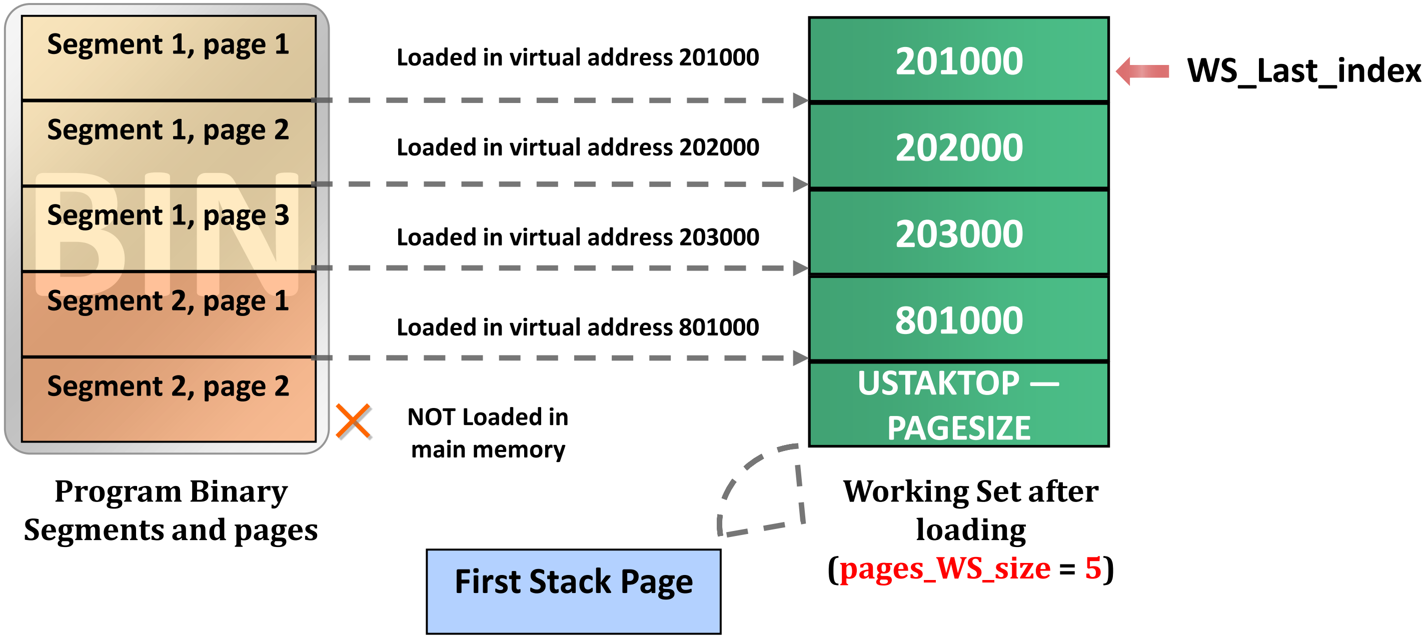
env_create( )

BIN

Set of pages

Required Tables

All Pages



Program Binary Segments and pages		Working Set after loading (pages_WS_size = 5)
Segment 1, page 1	Loaded in virtual address 201000	201000
Segment 1, page 2	Loaded in virtual address 202000	202000
Segment 1, page 3	Loaded in virtual address 203000	203000
Segment 2, page 1	Loaded in virtual address 801000	801000
Segment 2, page 2	NOT Loaded in main memory	USTAKTOP — PAGESIZE

Program Binary Segments and pages

Working Set after loading (pages_WS_size = 5)

Loading Program (env_create)

Three kernel dynamic allocations:

1. **create_user_page_WS():** should create new array for pages WS with the given size **[DONE]**
2. **create_user_directory():** should create new user directory **[DONE]**
3. **create_page_table():** should create new page table and link it to the directory. **[REQUIRED]**
   REMEMBER TO:
   1. clear all entries (as it may contain garbage data)
   2. clear the TLB cache (using "tlbflush()")

Project Features

[KERNEL]

1. **Kernel Heap:** dynamic allocation and free
   - **NEXT FIT** strategy
2. **Load and run** multiple user programs (***mostly DONE***)
3. **Page fault handler**
   - **MODIFIED CLOCK** replacement algorithm
4. **CPU Scheduling:** multi-level feedback queue

[USER]

1. **User Heap:** dynamic allocation and free
   - **NEXT FIT** strategy

Page Fault Handler

Modified Clock

Uses “use bit” & “modified bit”

4 states: (u, m)

- Not accessed recently, not modified (0, 0)
- Accessed recently, not modified (1, 0)
- Not accessed recently, modified (0, 1)
- Accessed recently, modified (1, 1)

BEST candidate: (0, 0)…

Modified Clock

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

Project Features

[KERNEL]

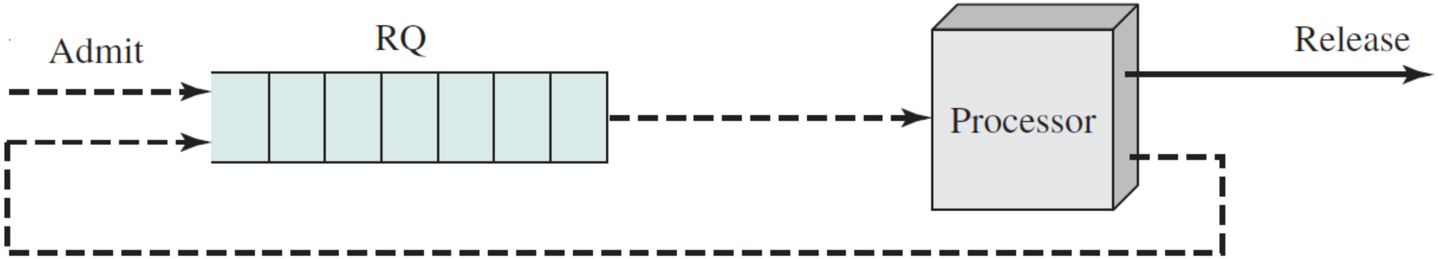
1. **Kernel Heap:** dynamic allocation and free
   - **NEXT FIT** strategy
2. **Load and run** multiple user programs (***mostly DONE***)
3. **Page fault handler**
   - **MODIFIED CLOCK** replacement algorithm
4. **CPU Scheduling:** multi-level feedback queue

[USER]

1. **User Heap:** dynamic allocation and free
   - **NEXT FIT** strategy

CPU Scheduling

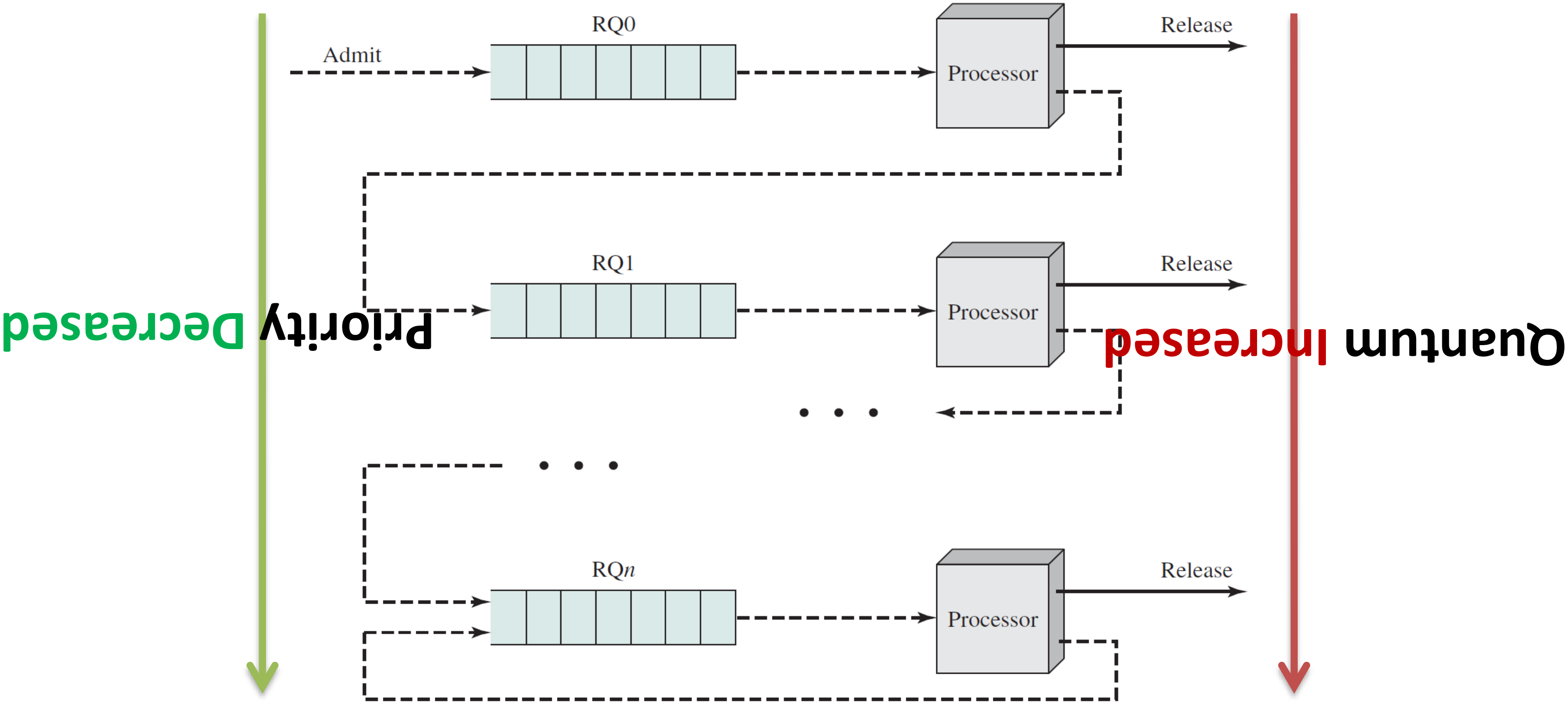
- Default: round robin method



- **Drawback**: favor processor-bound processes over I/O-bound processes, which results:
  1. in poor performance for I/O-bound processes,
  2. inefficient use of I/O devices,
  3. an increase in the variance of response time.

CPU Scheduling [MLFQ]

- **Solution**: multilevel feedback queue
  1. Penalize jobs that have been running longer
  2. Don’t know remaining time process needs to execute



CPU Scheduling [MLFQ]

- Given:
  1. Data Structures
  2. Queue Functions
- Your Task:
  1. Create the data structures
     1. Queues array
     2. Quantums array
  2. Handle the **scheduler**
     1. Place the current environment
     2. Select the next environment
     3. Set the proper quantum

CPU Scheduling [MLFQ]

- Data Structures

kern/sched.h

```
//[1] Ready queue(s) for the MLFQ or RR
struct Env_Queue *env_ready_queues;

//[2] Quantum(s) in ms for each level of the ready queue(s)
uint8 *quantums ;

//[3] Number of ready queue(s)
uint8 num_of_ready_queues ;
```

CPU Scheduling [MLFQ]

- Queue Functions (**DONE**)

kern/sched.c

```
void init_queue(struct Env_Queue* queue)

int queue_size(struct Env_Queue* queue)

void enqueue(struct Env_Queue* queue, struct Env* env)

struct Env* dequeue(struct Env_Queue* queue)

void remove_from_queue(struct Env_Queue* queue, struct Env* e)

struct Env* find_env_in_queue(struct Env_Queue* queue, uint32
envID)
```

Refer to **APPENDIX IV** in

Project Documentation for **Scheduler Functions**

CPU Scheduling [MLFQ]

- Given Function

kern/kclock.c

```
void kclock_set_quantum(uint8 quantum_in_ms)
```

(**DONE**)

1. Set the CPU clock by the given quantum
2. When this quantum is finished, a **H/W interrupt** is raised
3. The OS catches this interrupt and call **fos_scheduler()** to pick up the next environment

CPU Scheduling [MLFQ]

- Required Function

kern/sched.c

```
void sched_init_MLFQ(uint8 numOfLevels, uint8 *quantumPerLevel)
```

1. Create and initialize the data structures of the MLFQ:
    1. **num_of_ready_queues**
    2. Array of ready queues “**env_ready_queues**”
    3. Array of quantums "**quantums**"
2. Set the CPU quantum by the first level one

CPU Scheduling [MLFQ]

- Required Function

kern/sched.c

```
void fos_scheduler(void)
```

1. Check the existence of the **current environment** and place it in the **suitable queue**
2. **Search the queues** according to their priorities (first is highest)
3. If environment is found:
    1. Set the "**next_env**" by the found environment
    2. Set the **CPU clock** by the quantum of the selected level

Project Features

[KERNEL]

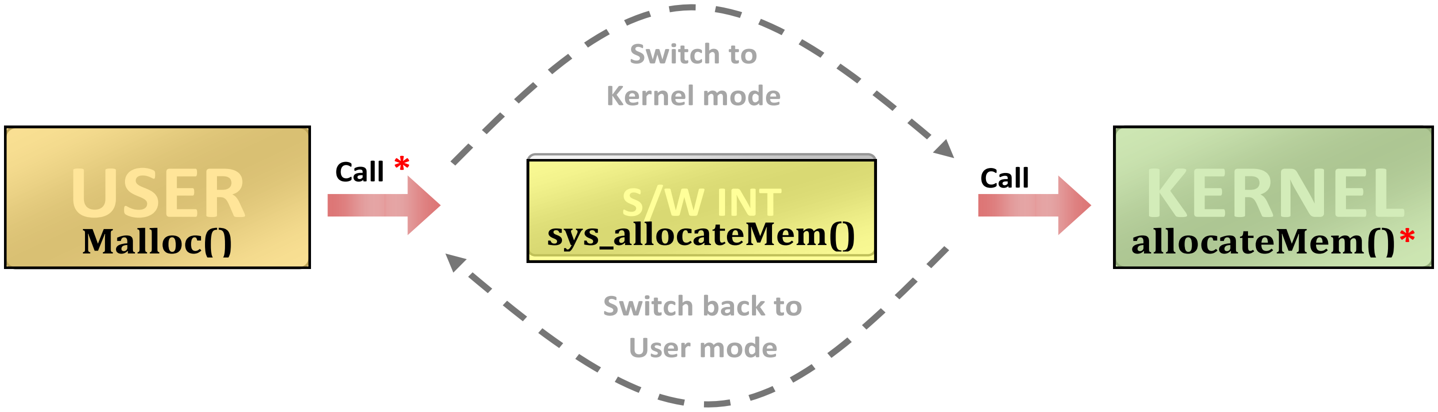
1. **Kernel Heap:** dynamic allocation and free
   - **BEST FIT** strategy
2. **Load and run** multiple user programs (***mostly DONE***)
3. **Page fault handler** during execution
   - **MODIFIED CLOCK** replacement algorithm
4. **CPU Scheduling: multi-level feedback queue**

[USER]

1. **User Heap:** dynamic allocation and free
   - **NEXT FIT** strategy

[USER] Project Features

- **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

Dynamic allocation/Deallocation [**malloc() / free()**]

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

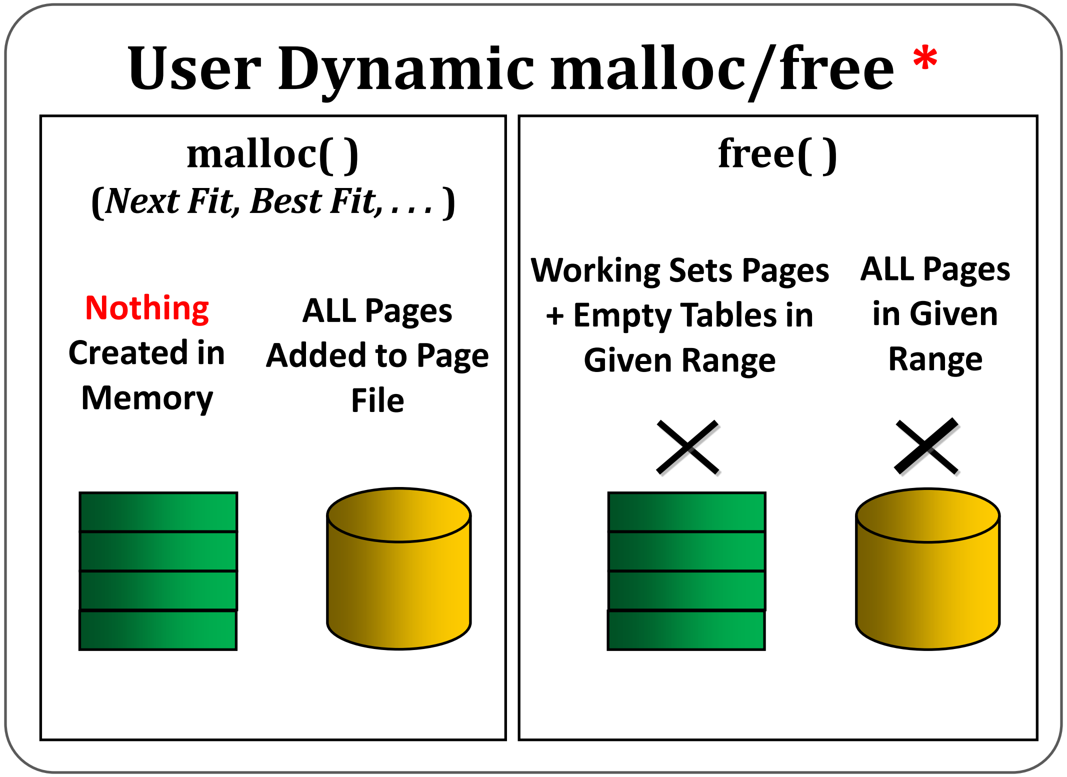
Dynamic allocation/Deallocation [malloc() / free()]

- **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"

Dynamic allocation/Deallocation [malloc() / free()]

- **De-allocation (free)**
  - **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"

Dynamic allocation/Deallocation [malloc() / free()]



User Dynamic malloc/free *

malloc( )

(*Next Fit, Best Fit, . . .*)

Nothing Created in Memory

ALL Pages Added to Page File

free( )

Working Sets Pages + Empty Tables in Given Range

ALL Pages in Given Range

Dynamic allocation/Deallocation [malloc() / free()]

- **Required Functions**

[USER SIDE] Lib/uheap.c
[KERNEL SIDE] kern/memory_manager.c

```
void* malloc(uint32 size)
```

1. Implement NEXTT FIT strategy to search the heap for suitable space to the required allocation size (space should be on **4 KB BOUNDARY**)
2. if no suitable space found, return NULL, else,
3. Call **sys_allocateMem** to invoke the Kernel for allocation
4. Return pointer containing the virtual address of allocated space

```
void allocateMem(struct Env* e,uint32 virtual_address,
uint32 size)
```

- allocate ALL pages of the required size in the **Page File** (Don’t allocate any frame in the RAM)

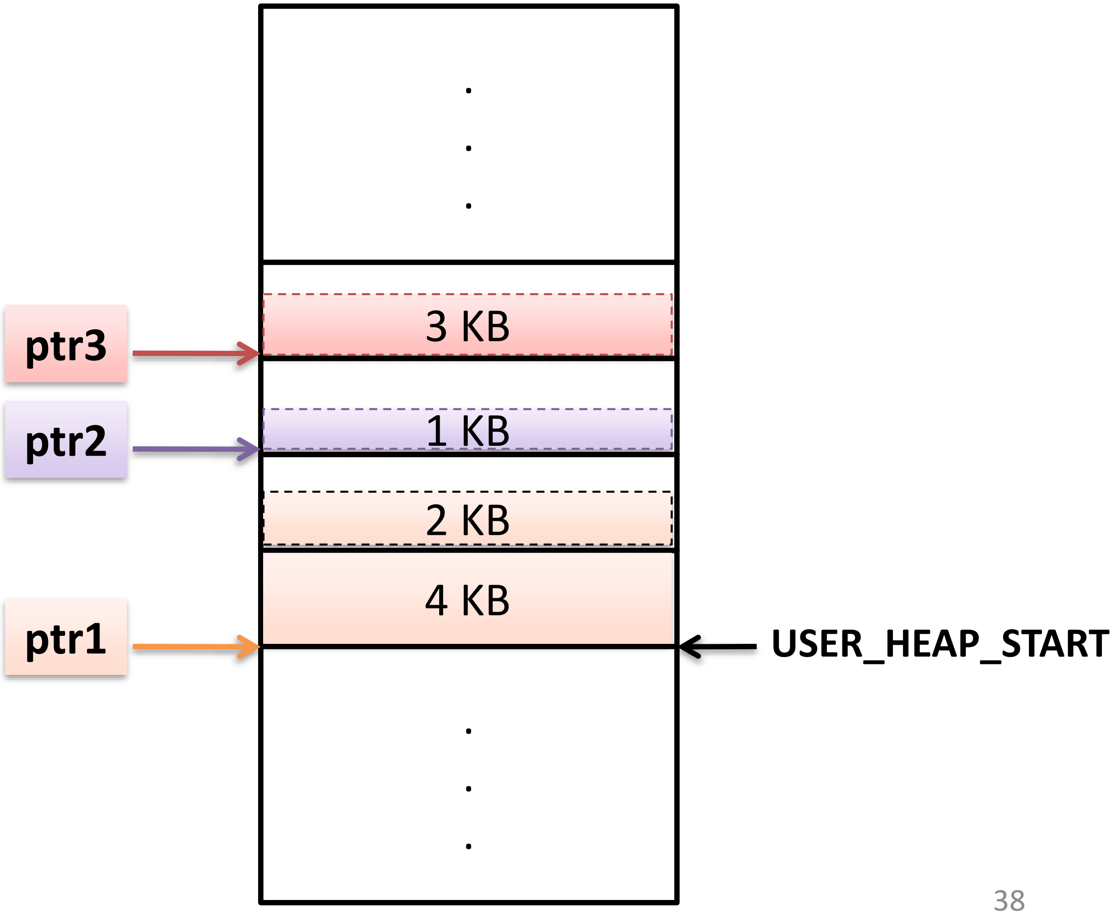
Dynamic allocation/Deallocation [malloc() / free()]

- **Allocate pages on 4KB granularity**

ptr3 = malloc (3 KB)

ptr2 = malloc (1 KB)

ptr1 = malloc (6 KB)



ptr3

ptr2

ptr1

3 KB

1 KB

2 KB

4 KB

USER_HEAP_START

Dynamic allocation/Deallocation [malloc() / free()]

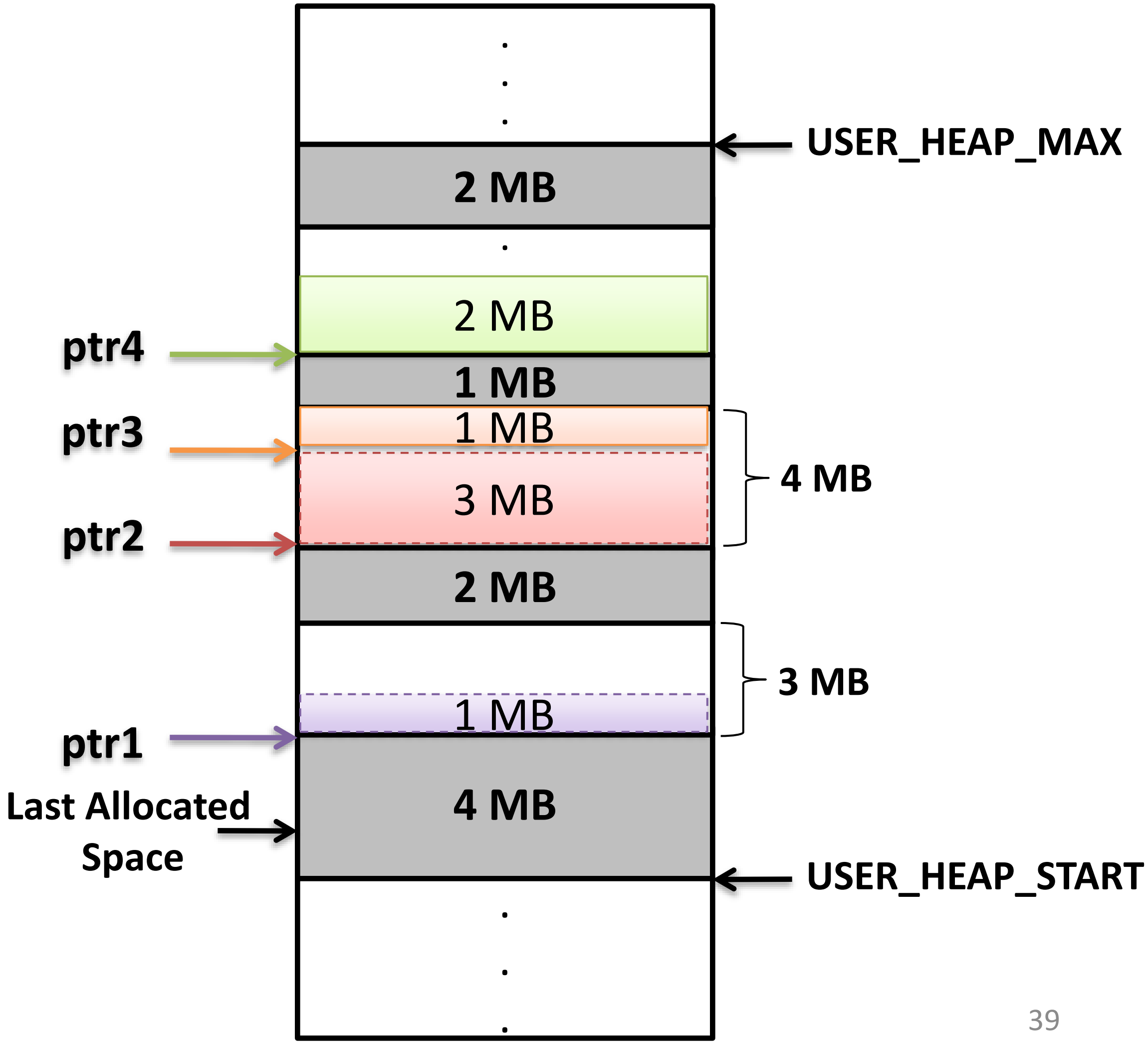
- **NEXT FIT Strategy**

ptr4 = malloc (2 MB)

ptr3 = malloc (1 MB)

ptr2 = malloc (3 MB)

ptr1 = malloc (1 MB)



Dynamic allocation/Deallocation [malloc() / free()]

- **Required Functions**

[USER SIDE] Lib/uheap.c
[KERNEL SIDE] kern/memory_manager.c

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

1. Find the allocated size of the given virtual_address
2. Frees this allocation from the user Heap
3. Call “**sys_freeMem**” to free the allocation from the memory & page file

```
void freeMem(struct Env* e, uint32 virtual_address, uint32
size)
```

1. Free ALL pages of the given range from the Page File
2. Free ONLY pages that are resident in the working set from the memory
3. Removes ONLY the empty page tables (i.e. no pages are mapped in it)

Project Features

[KERNEL]

1. **Kernel Heap:** dynamic allocation and free
   - **BEST FIT** strategy
2. **Load and run** multiple user programs **(*mostly DONE*)**
3. **Page fault handler** during execution
   - **MODIFIED CLOCK** replacement algorithm
4. **CPU Scheduling: multi-level feedback queue**

[USER]

1. **User Heap:** dynamic allocation and free
   - **NEXT FIT** strategy

BONUSES

Bonuses

1. Strategies for Kernel Dynamic Allocation

- Beside the NEXT FIT strategy, implement the BEST FIT one to find the suitable space for allocation.
- Compare their performance on one or more programs.

Bonuses

2. Free the entire environment (exit)

1. All pages in the page working set
2. The working set itself
3. All page tables in the entire user virtual memory
4. Directory table
5. All pages from page file, this code ***is already*** written for you ☺

Bonuses

3. User Realloc

- Attempts to resize the allocated space at given virtual address to "new size" bytes, possibly moving it in the heap.
    - If successful, returns the new virtual address, in which case the old virtual address must no longer be accessed.
    - On failure, returns a null pointer, and the old virtual address remains valid.
- A call with virtual_address = null is equivalent to malloc()
- A call with new_size = zero is equivalent to free()

Bonuses

4. Add “Program Priority” Feature to FOS

- 5 different priorities can be assigned to any environment:
  1. Low
  2. Below Normal
  3. Normal **[default]**
  4. Above Normal
  5. High
- Kernel can set/change the priority of any environment
- Priority affects the working set (WS) size, as follows:

Priority	Effect on WS Size
Low	decrease WS size by its half **IMMEDIATELY** by removing half of it using replacement strategy
Below Normal	decrease WS size by its half **ONLY** when half of it become empty
Normal	**no change** in the original WS size
Above Normal	double the WS size when it becomes full (**1 time only**)
High	double the WS size **EACH TIME** it becomes full (until reaching half the RAM size)

CHALLENGES

CHALLENGES

FIRST: Stack De-Allocation

- To avoid the leak in the stack area, remove the UN-NEEDED stack pages from both memory and its copy on the page file as well.
- Refer to documentation for more details

CHALLENGES

SECOND: System Hibernate

- Add a command to hibernate the system by:
  1. Saving the status of:
     - Main memory
     - Page file
  2. Close the system
- When opened again, without recompilation, the system is restored.

PROJECT QUICK GUIDE

Startup Code

FOS_PROJECT_2022_Template.Zip

Follow these steps to import the project folder into the eclipse

ALL Required Functions

Tasks

1. **Kernel Heap**

MAIN Functions	
kmalloc	
kfree	
kheap_virtual_address	
kheap_physical_address	
create_page_table	

1. **Page Fault Handler** [2 cases]
2. **Scheduler**
    1. Scheduler_init()
    2. Fos_scheduler()

ALL Required Functions

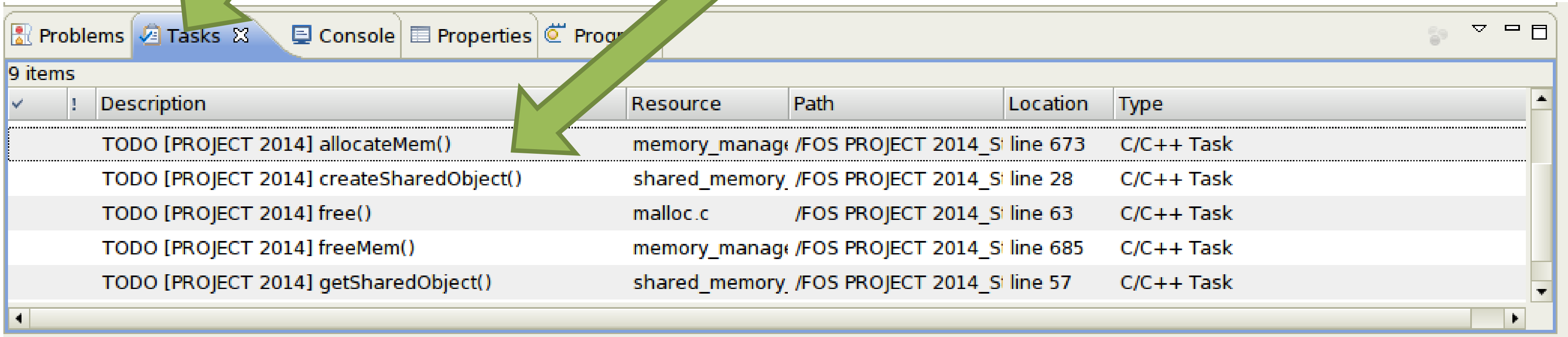
1. **User Heap**
    1. malloc
    2. allocateMem
    3. free
    4. freeMem

Where should I write the Code?

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

[1] Click on "Tasks" Tab

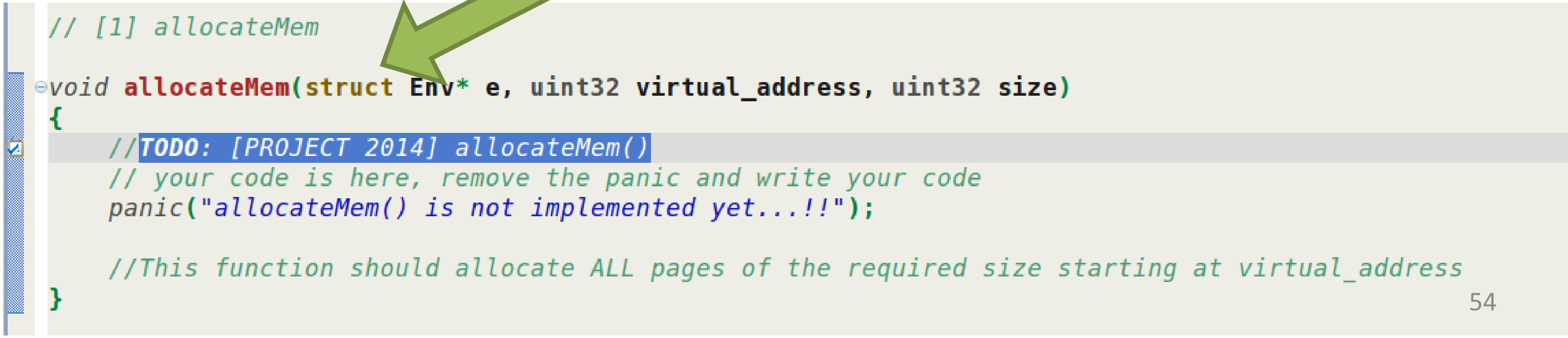
[2] Double Click on the required function



9 items

✓	!	Description	Resource	Path	Location	Type
		TODO [PROJECT 2014] allocateMem()	memory_manag	/FOS PROJECT 2014_S	line 673	C/C++ Task
		TODO [PROJECT 2014] createSharedObject()	shared_memory_	/FOS PROJECT 2014_S	line 28	C/C++ Task
		TODO [PROJECT 2014] free()	malloc.c	/FOS PROJECT 2014_S	line 63	C/C++ Task
		TODO [PROJECT 2014] freeMem()	memory_manag	/FOS PROJECT 2014_S	line 685	C/C++ Task
		TODO [PROJECT 2014] getSharedObject()	shared_memory_	/FOS PROJECT 2014_S	line 57	C/C++ Task

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

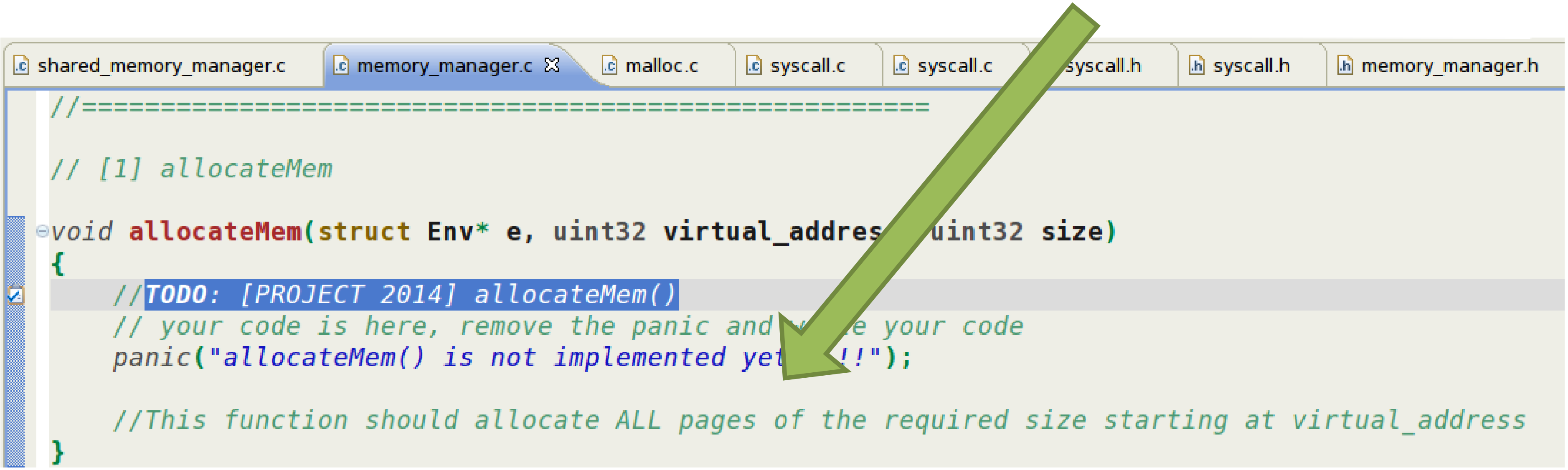


```
// [1] allocateMem

void allocateMem(struct Env* e, uint32 virtual_address, uint32 size)
{
    //TODO: [PROJECT 2014] allocateMem()
    // your code is here, remove the panic and write your code
    panic("allocateMem() is not implemented yet...!!");

    //This function should allocate ALL pages of the required size starting at virtual_address
}
```

What about the steps?

You’ll find it inside each function

Detailed Steps



```
//=======================================================

// [1] allocateMem

void allocateMem(struct Env* e, uint32 virtual_address, uint32 size)
{
    //TODO: [PROJECT 2014] allocateMem()
    // your code is here, remove the panic and write your code
    panic("allocateMem() is not implemented yet !!");

    //This function should allocate ALL pages of the required size starting at virtual_address
}
```

How I ensure it’s correct?

- There’re **test programs** that test
  - Each function separately
  - Entire project
- Just run the test program & it tell you if it succeed or not

Helper Functions

- Set of **ready-made functions** are available to help you when writing your solution.
- **Detailed description** can be found in **documentation**

Delivery

- **Dropox-based… Fully automated**
- **Similar test cases** will be used to evaluate your solution
- Each case **is binary**: success (1) or not (0)
- **Make sure** they run correctly before you deliver isA
- **Delivery Dates:**
  - **SUN** of Lab Exam Week
- **Support Dates:**
  - *WEEKLY Office Hours*

Thank you for your care…

Enjoy **making** your **own FOS** ☺