



Faculty of Computer and Information Sciences

Ain Shams University

CHP – Second Semester

2022 - 2023

Operating Systems

FOS KERNEL MEMORY MANAGEMENT PROJECT

Dr. Hanan Hindy

Contents

INTRODUCTION	5
What's new?!.....	5
Project Specifications	5
New Concepts	6
FIRST: Working Sets.....	6
SECOND: Page File	7
THIRD: Page Buffering.....	7
FOURTH: System Calls.....	8
OVERALL VIEW	10
DETAILS	11
FIRST: Kernel Heap	11
SECOND: Fault Handler	13
THIRD: User Heap → Dynamic Allocation and Free.....	16
TESTING	17
A- How to test your project.....	17
BONUS:.....	18
FIRST: User Realloc.....	18
SECOND: Free the entire environment (exit).....	18
APPENDIX I: PAGE FILE HELPER FUNCTIONS.....	19
Pages Functions.....	19
Add a new environment page to the page file.....	19
Read an environment page from the page file to the main memory	19
Update certain environment page in the page file by contents from the main memory	20
APPENDIX II: WORKING SET STRUCTURE & HELPER FUNCTIONS	20
Working Set Structure.....	20
Working Set Functions	21
Get Working Set Current Size.....	21

Set Virtual Address of Page in Working Set.....	21
Clear Entry in Working Set	22
Check If Working Set Entry is Empty	22
APPENDIX III: MANIPULATING PERMISSIONS IN PAGE TABLES AND DIRECTORY	23
Permissions in Page Table.....	23
Set Page Permission	23
Get Page Permission.....	23
Clear Page Table Entry	24
Permissions in Page Directory.....	24
Clear Page Dir Entry.....	24
Check if a Table is Used	24
Set a Table to be Unused	25
APPENDIX IV: PAGE BUFFERING	26
Structures	26
Functions	26
Add Frame to the Tail of a Buffered List	26
Remove Frame from a Buffered List	27
Get Current Size of a Buffered List	27
Get Maximum Length of the Modified Buffered List	27
Iterate on ALL Elements of a Buffered List.....	28
APPENDIX V: MODIFIED CLOCK REPLACEMENT ALGORITHM	29
Algorithm Description	29
Example	29
APPENDIX VI: COMMAND PROMPT.....	30
NEW Command Prompt Features	30
First: DOSKEY	30
Second: TAB Auto-Complete.....	30
APPENDIX VII: BASIC AND HELPER MEMORY MANAGEMENT FUNCTIONS	30
Basic Functions.....	30

LOGISTICS: refer to [power point presentation](#) in the project materials

Delivery:

1. **Dropbox-based**
2. **Milestone 1:** THU of Week#9 (13 APR till 11:59 PM) → kernel heap functions
3. **Milestone 2:** THU of Week#13 (11 MAY till 10:00 PM) → fault handler, user heap

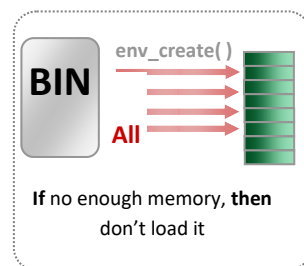
For Milestone1 & 2:

- It's **FINAL** delivery
- **MUST** deliver the required tasks and **ENSURE** they're worked correctly.
- **Code:** Make sure you are working on the **FOS_PROJECT_2023_template.zip** provided to you; Follow [these steps](#) to import the project folder into the eclipse

Introduction

What's new?!

Previously: all segments of the program binary plus the stack page should be loaded in the main memory. If there's no enough memory, the program will not be loaded. See the following figure:

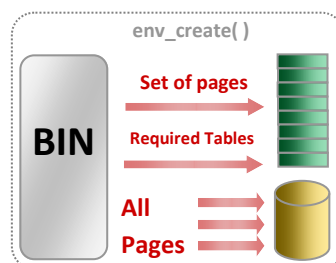


But, wait a minute...

This means that you may not be able to run one of your programs if there's no enough main memory for it!! This is not the case in real OS!! In windows for example, you can run any program you want regardless the size of main memory... do you know WHY?!

YES... it uses part of secondary memory as a virtual memory. So, the loading of the program will be distributed between the main memory and secondary memory.

NOW in the task: when loading a program, only part of it will be loaded in the main memory while the whole program will be loaded on the secondary memory (H.D.D.). See the following figure:



This means that a “Page Fault” can occur during the program run. (i.e. an exception thrown by the processor to indicate that a page is not present in main memory). The kernel should handle it by loading the faulted page back to the main memory from the secondary memory (H.D.D.).

Project Specifications

In the light of the studied memory management system, you are required add new features for your FOS memory manager, these new features are:

- 1- **Kernel heap:** allow the kernel to dynamically allocate and free memory space at run-time (for user directory, page tables ...etc.)
- 2- Handle **page faults** during execution by applying **MODIFIED CLOCK replacement algorithm**.
- 3- **User heap:** Allow user program to dynamically allocate and free memory space at run-time (i.e., malloc and free functions) by applying **BEST FIT strategy**.

For loading only part of a program in main memory, we use the **working set** concept; the working set is the set of all pages loaded in main memory of the program at run time at any instant of time.

Each **program environment** is modified to hold its working sets information. The working sets are a **FIXED** size array of virtual addresses corresponding to pages that are loaded in main memory.

The virtual space of any loaded user application is as described in lab 6, see **Figure 1**.

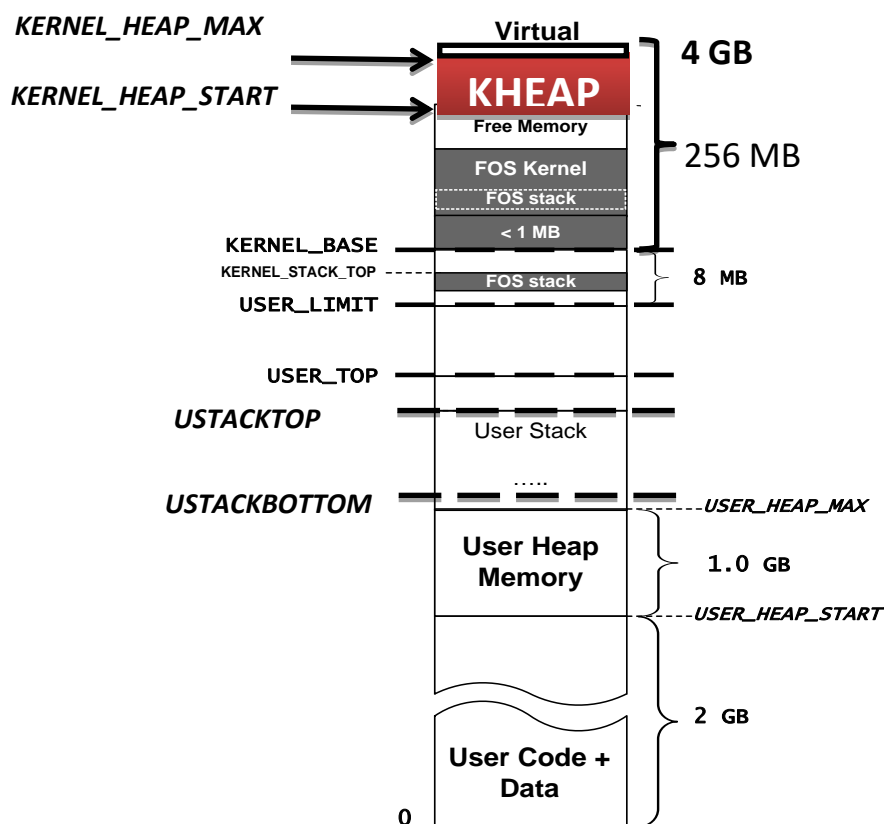


Figure 1: Virtual space layout of single user loaded program

There are three important concepts you need to understand in order to implement the project; these concepts are the **Working Set**, **Page File** and **System Calls**.

New Concepts

FIRST: Working Sets

We have previously defined the working set as the set of all pages loaded in main memory of the program at run time which is implemented as a **FIXED** size array of virtual addresses corresponding to pages that are loaded in main memory.

See [Appendix II](#) for description about the working sets data structure and helper functions.

Although it's a fixed size array, but its size is different from one program to another. Its max size is specified during the `env_create()` and is set inside the "struct Env", let's say **N** pages, when the program needs memory (*either when the program is being loaded or by dynamic allocation during the program run*) FOS must allow maximum **N** pages for the program in main memory.

So, when page fault occurs during the program execution, the page should be loaded from the secondary memory (the **PAGE FILE** as described later) to the **WORKING SET** of the program. If the working set is complete, then one of the program environment pages from the working set should be replaced with the new page. This is called **LOCAL** replacement since each program should replace one of its own loaded pages. This means that our working sets are **FIXED size with LOCAL replacement facility**.

FOS must maintain the working sets during the run time of the program, when new pages from PAGE FILE are loaded to main memory, the working set must be updated.

The initialization of the working set by the set of pages during the program loading is already implemented for you in "`env_create()`", and you will be responsible for maintaining the working set during the run time of the program.

Therefore, the working sets of any loaded program must always contain the correct information about the pages loaded in main memory.

SECOND: Page File

The page file is an area in the secondary memory (H.D.D.) that is used for saving and loading program pages during runtime. Thus, for each running program, there is a storage space in the page file for **ALL** pages needed by the program, this means that user code, data, stack and heap sections are **ALL** written in page file. (Remember that not all these pages are in main memory, only the working set). You might wonder why we need to keep all pages of the program in secondary memory during run!!

The reason for this is to have a copy of each page in the page file. So, we don't need to write back each swapped out page to the page file. Only **MODIFIED** pages are written back to the page file.

But wait a minute... Did we have a file manager in FOS?!!

NO...! Don't panic, we wrote some helper functions for you that allow us to deal with the page file. *See [Appendix I](#) for description about these helper functions.*

*The loading of **ALL** program binary segments plus the stack page to the page file is already implemented for you in "`env_create()`", you will need to maintain the page file in the rest of the project.*

THIRD: Page Buffering

An interesting strategy that can improve paging performance is page buffering. A replaced page is not lost but rather is assigned to one of two lists:

1. the free frame list if the page has not been modified, or
2. the modified page list if it has.

See [Appendix IV](#) for description about the page buffering structures and helper functions.

Note that the page is not physically moved about in main memory; instead, the PRESENT bit for this page is set to 0 and its "Frame_Info" is placed in the tail of either the free or modified list.

Buffering a Page:

To indicate whether or not this page is buffered in the memory, we use the following two values:

- 1- **BUFFERED bit**: one of the available bits in the page table is used to indicate whether the frame of certain virtual address is buffered or not
- 2- **isBuffered**: Boolean defined inside the "Frame_Info" structure to indicate whether the corresponding frame is buffered or not.

IMPORTANT NOTE: These two values should be set/reset together inside the code

If we buffer the page, then we should set its BUFFERED bit to 1 inside the page table together with the "isBuffered" variable inside its "Frame_Info" structure. (Refer to [Appendix IV](#) for more details)

When a page is to be allocated, the frame at the head of the free frame list is used, destroying the page that was there.

When an unmodified page is to be replaced, it remains in memory and its "Frame_info" is added to the tail of the free frame list. Similarly, when a modified page is to be written out and replaced, its "Frame_info" is added to the tail of the modified page list.

Freeing the Buffer

The pages on the modified list can periodically be written out in batches when their count reaches "`MAX_MODIFIED_LIST_COUNT`" (defined in "`inc/environment_definitions.h`") and moved to the tail of free frame list. This page (on the free frame list) is then either reclaimed if it is referenced or lost when its frame is assigned to another page.

There are two important aspects of these maneuvers

1. The page to be replaced remains in memory. Thus, if the process references that page, it is returned to the working set of that process at little cost (by setting its PRESENT bit to 1).
2. Modified pages are written out in clusters rather than one at a time. This significantly reduces the number of I/O operations and therefore the amount of disk access time.

FOURTH: System Calls

- You will need to implement a dynamic allocation function (and a free function) for your user programs to make runtime allocations (and frees).
- The user should call the kernel to allocate/free memory for it.
- But wait a minute...!! remember that the user code is not the kernel (i.e., when a user code is executing, the processor runs **user mode** (less privileged mode), and to execute kernel code the processor needs to go from user mode to **kernel mode**)
- The switch from user mode to kernel mode is done by a **software interrupt** called "**System Call**".
- All you need to do is to call a function prefixed "**sys_**" from your user code to call the kernel code that does the job, (e.g. from user function `malloc()`, call `sys_allocateMem()` which then will call kernel function `allocateMem()` to allocate memory for the user) as shown in figure:

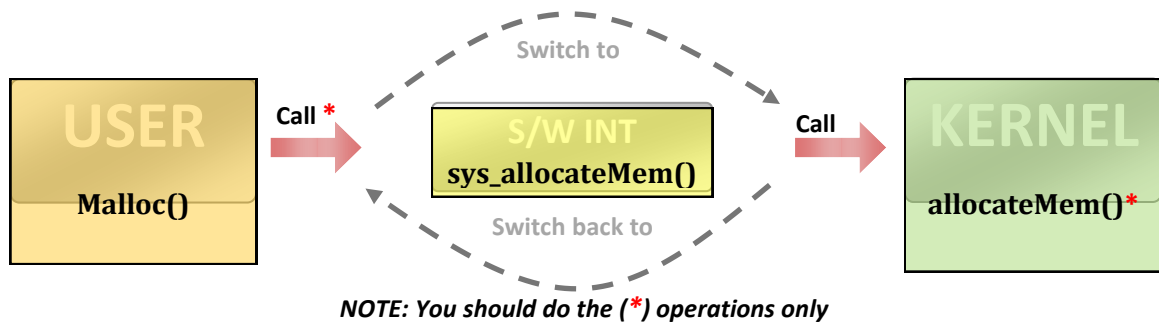


Figure 2: Sequence diagram of dynamic allocation using malloc()

Overall View

The following figure shows an overall view of all project components together with the interaction between them. The components marked with (*) should be written by you, the unmarked components are already implemented.

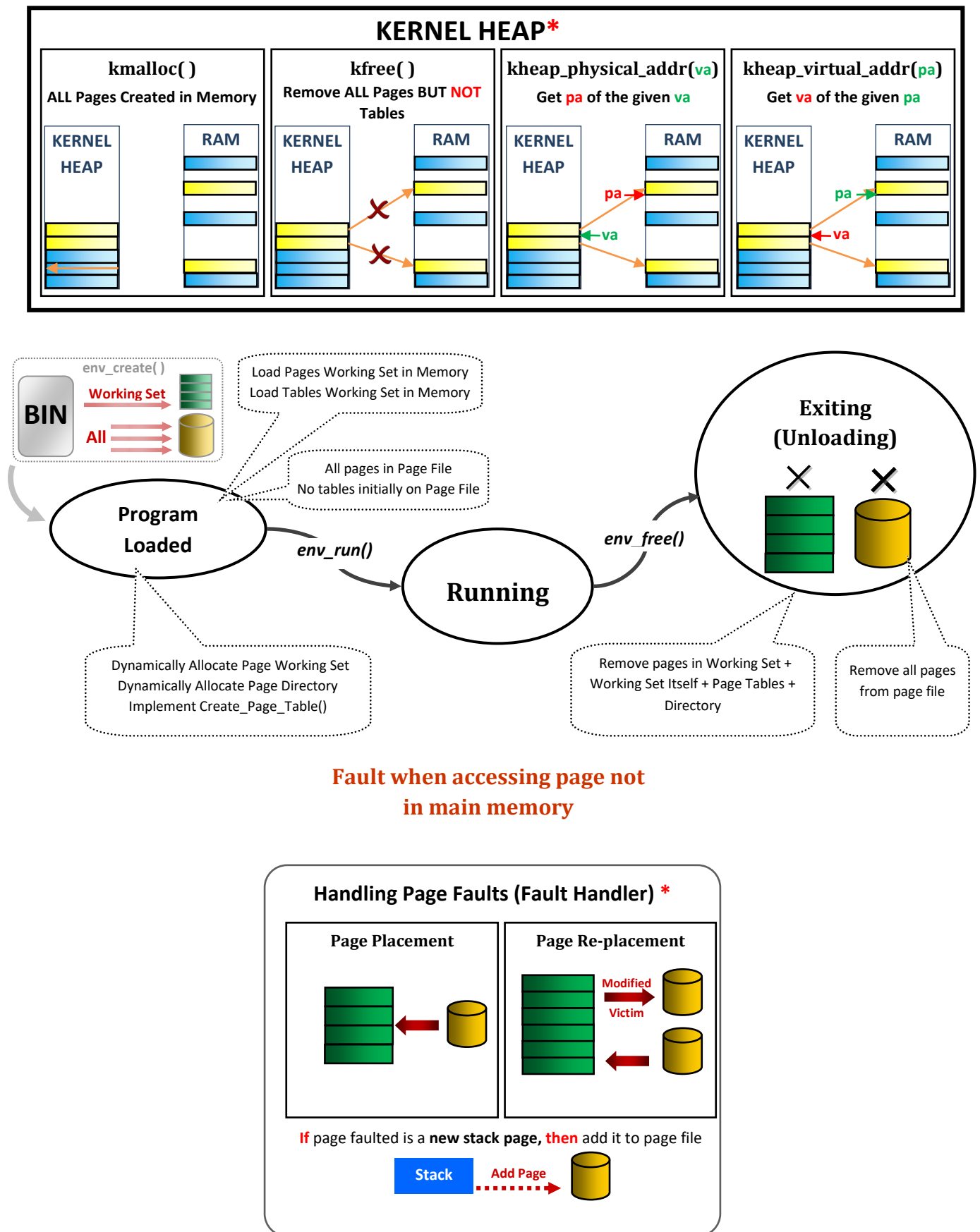


Figure 3: Interaction among components in project

Details

FIRST: Kernel Heap

Problem

The kernel virtual space [KERNEL_BASE, 4 GB) is **one-to-one** mapped to the physical memory [0, 256 MB), as shown in Figure 4. This limits the physical memory area for the kernel to 256 MB only. In other words, the kernel code can't use any physical memory after the 256 MB, as shown in Figure 5.

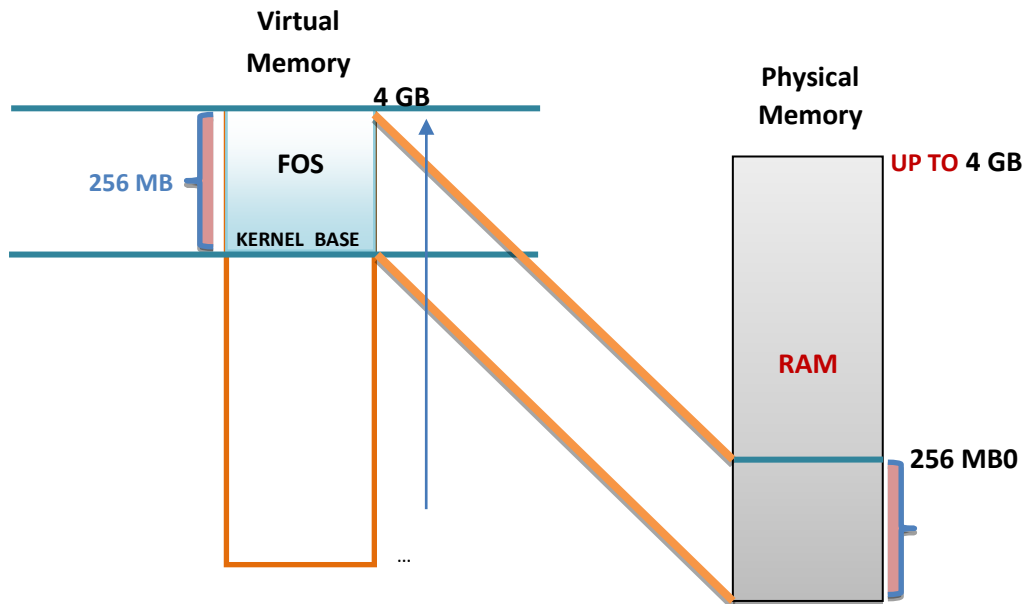


Figure 4: Mapping of the FOS VIRTUAL memory to the PHYSICAL memory [32-bit Mode]

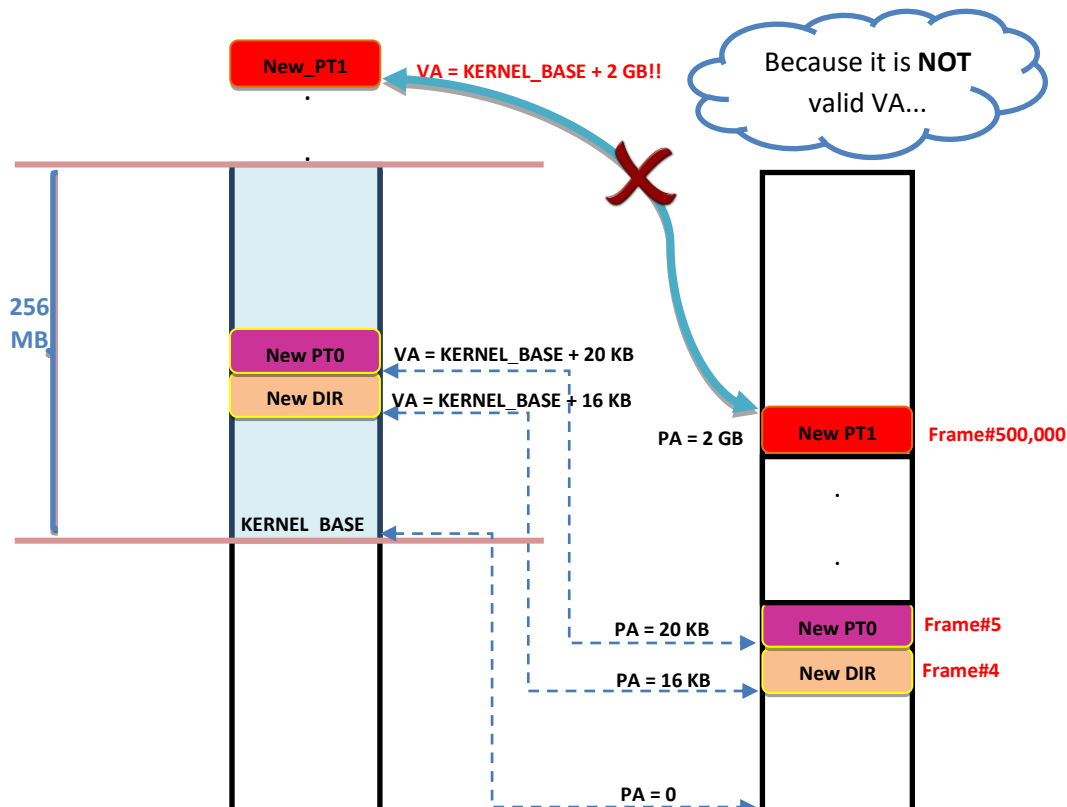


Figure 5: PROBLEM of ONE-to-ONE mappings between the FOS VIRTUAL memory and its corresponding frames in the PHYSICAL memory

Solution

Replace the one-to-one mapping to an ordinary mapping as we did in the allocation of the user's space. This allows the kernel to allocate frames anywhere and reach it wherever it is allocated by saving its frame number in the page table of the kernel, as shown in **Figure 6**.

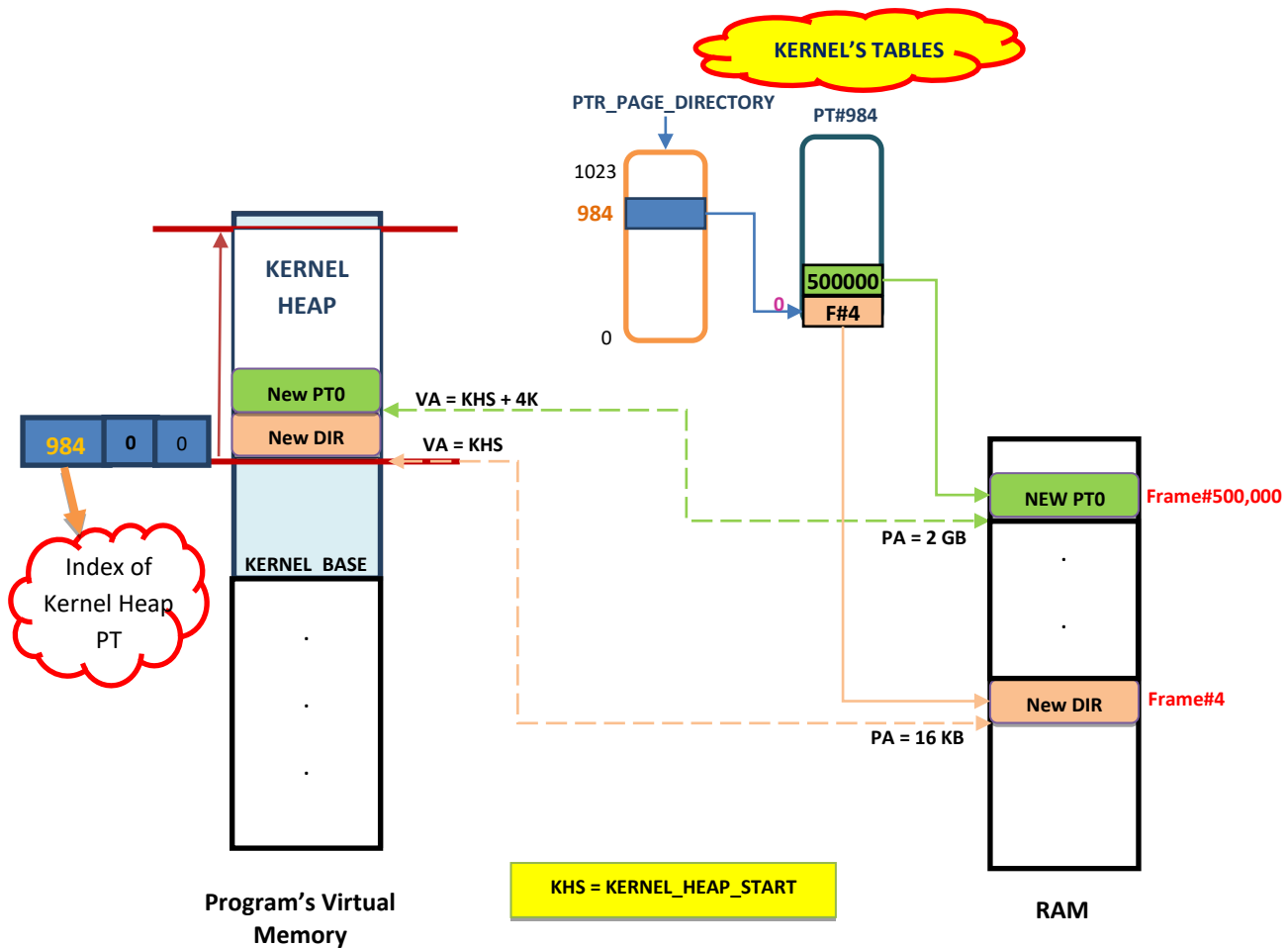


Figure 6: New Model to map kernel's VA of an allocated page in the KERNEL HEAP to pa

However, since the Kernel virtual space should be shared in the virtual space of all user programs, ALL page tables of the kernel virtual space (from table#960 to table#1023) are already created and linked to the Kernel directory. These tables **SHOULD NOT be removed** for any reason until the FOS is terminated.

You need to fill the following functions that serve the **Kernel's Heap**:

IMPORTANT: Refer to the [ppt](#) inside the project materials for more details and examples

1. **Kmalloc():** dynamically allocate space size using **FIRST FIT** Strategy and map it as shown in **Figure 6**. It should work as follows:

Search for suitable place using the **FIRST FIT** strategy, if found:

1. Allocate free frame(s) for the required size
2. Map the new frame(s) with the virtual page(s) starting from the found location.

Else, return NULL

2. **Kfree():** delete a previously allocated space by removing all its pages from the Kernel Heap

Q) After calling kfree, if the page table becomes empty and all the entries within it are unmapped, i.e. cleared, **shall we delete this page table?**

A) No, take care this table is one of the kernel's table that are shared with ALL loaded user programs.

Q) Do you remember the first step we did in env_create to start creating a new environment for a program to be loaded in memory?

A) Yes we create new virtual by allocating a frame for the new directory then we copy the kernel's directory in the new directory to share the FOS kernel part.

For this reason, if we take the decision to delete the page table, we shall delete it from all other programs and vice versa if it is allocated again at one process, it shall be created for all processes...

So, take care we delete and un-map PAGES ONLY not TABLES.

- Unmap each page of the previously allocated space at the given address by clearing its entry in the page table and free its frame.
- **DO NOT** remove any table

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

- For example, in **Figure 6**, if we call kheap_virtual_address() for physical address 16 KB, it should return KHS (KERNEL_HEAP_START).

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

- For example, in **Figure 6**, if we call kheap_physical_address() for virtual address KHS + 4 KB, it should return 2 GB.

SECOND: Fault Handler

IMPORTANT: Refer to the [ppt](#) inside the project materials for more details and examples

In function fault_handler(), " kern/trap.c"

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

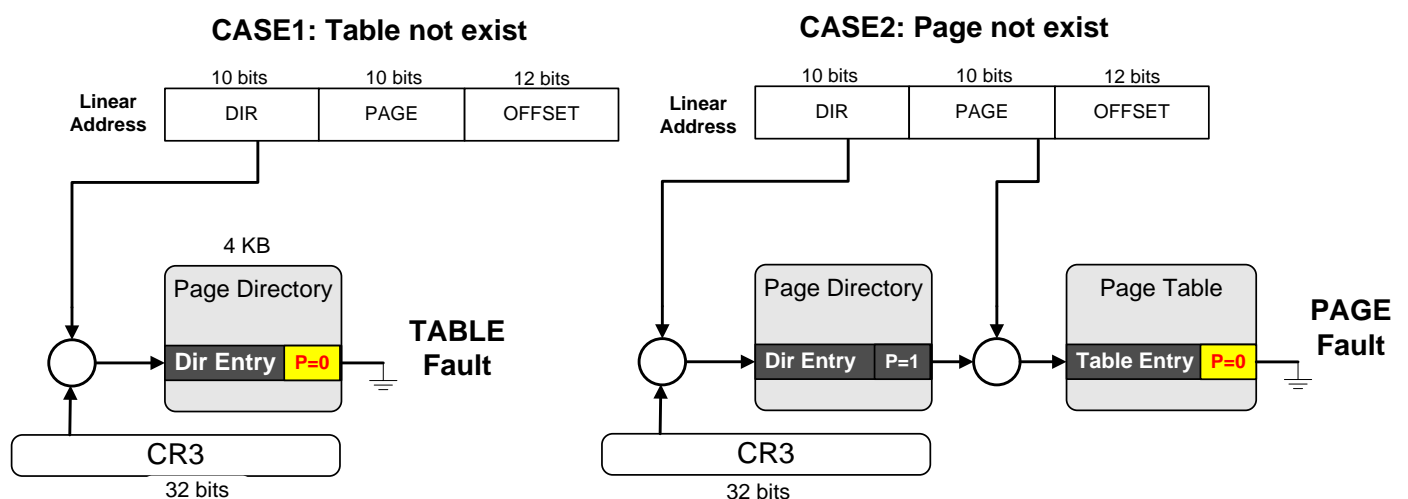


Figure 7: Fault types (table fault and page fault)

- The first case (Table Fault) is **already handled** for you by allocating a new page table if it not exists
- You **should handle** the page fault in FOS kernel by:
 - a. Allocate a new page for this faulted page in the main memory
 - b. Loading the faulted page back to main memory from page file if the page exists. Otherwise (i.e. new page), add it to the page file (for new stack pages only).
- You can handle the page fault in **function "page_fault_handler()"** in "trap.c".
- In **replacement** part, it's required to apply **MODIFIED CLOCK** algorithm.
You need to check which algorithm is currently selected using the given functions, as follows

```
//Page Replacement  
If isPageReplacmentAlgorithmModifiedCLOCK()  
    Apply MODIFIED CLOCK algorithm to choose the victim
```

- **PAGE FAULT HANDLER** should work as follows:

1. if the size of the page working set < **PAGE_WS_MAX_SIZE** then do

Placement:

1. check if required page is buffered (see [Appendix III](#)), if true, do the following:
 1. Just reclaim the page by setting PRESENT bit to 1, and BUFFERED bit to 0, and set its *frame_info->isBuffered* to 0. (See [Appendix III](#) and [Appendix IV](#))
 2. Remove its Frame_Info from: "modified_frame_list" if the page was modified (MODIFIED bit = 1), or from "free_frame_list" if the page was not modified (MODIFIED bit = 0) (see [Appendix III](#) and [Appendix IV](#))
 - else
 1. allocate and map a frame 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 **CHECK if it is a stack page**. If so this means that it is a new stack page, add a **new empty page** with this faulted address to page file (refer to [Appendix I](#) and [Appendix II](#)), else panic with invalid virtual address.
 2. update the page working set.
- else, do

Replacement:

3. implement the **modified clock** algorithm to find victim virtual address from page working set to replace (refer to [Appendix V](#) to see how the modified clock works)
 4. for the victim page:
 - Prepare its **Frame_Info** by
 - i. flagging it as buffered,
 - ii. setting the environment inside it,
 - iii. setting the victim virtual address inside it.
 - Set the BUFFERED bit to 1 in the victim page table.
 - Set the PRESENT bit to 0 in the victim page table.
 5. If the victim page was not modified, then:
 - Add the victim frame to "free_frame_list" at the tail (see [Appendix IV](#))
- Else
- Add the victim frame to "modified_frame_list" at the tail (see [Appendix IV](#))
Then, check the count of modified pages in "modified_frame_list" (see [Appendix IV](#)), if it reaches *its maximum capacity* (check this by calling *getModifiedBufferLength()*), do the following:
For each modified frames in this list (either belong to this environment or other environments), do (see [Appendix IV](#))
 - . update its page in page file (see [Appendix I](#)),
 - a. set the MODIFIED bit of its page in its page table to 0 (see [Appendix III](#))
 - b. add it to the tail of the "free_frame_list".

2. Apply **Placement** steps that are described before.

- Refer to helper functions to deal with flags of Page Table entries ([Appendix III](#))
- Refer to the basic and helper memory manager functions ([Appendix VII](#))

THIRD: User Heap → Dynamic Allocation and Free

IMPORTANT: Refer to the [ppt](#) inside the project materials for more details and examples

- You should implement function “**malloc()**” (allocates user memory) and “**free()**” (frees user memory) functions in the USER side “**lib/uheap.c**”
- You should use [THIRD: Page Buffering](#)
- You should use [FOURTH: System Calls](#) to switch from user to kernel.
- Your kernel code will be in “**allocateMem()**” and “**freeMem()**” functions in “**memory_manager.c**”

Dynamic Allocation

User Side (malloc)

1. Implement BEST 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.

You need to check which strategy is currently selected using the given functions.

[`sys_isUHeapPlacementStrategyBESTFIT()` ...]

Kernel Side (allocateMem)

In `allocateMem()`, all pages you allocate will **not be added** to the working set (not exist in main memory). Instead, they **should be added** to the page file, so that when this page is accessed, a page fault will load it to memory.

Dynamic De-allocation

User Side (free)

1. Frees the allocation of the given virtual address in the user Heap
2. Need to call “`sys_freeMem`” inside.

Kernel Side (`_freeMem_with_buffering`)

1. **For each BUFFERED** page in the given range inside the **given environment**
 - a) If the buffered page is MODIFIED, then
 - Get its “**Frame_info**” pointer (see [Appendix VII](#))
 - Remove it from the “**modified_frame_list**”
 - Flag it as a free frame (setting **isBuffered** to 0, **environment** to NULL)
 - Add it to the head of “**free_frame_list**” (by calling `free_frame()`)
 - b) Else
 - Just flag its “**Frame_Info**” as free frame (setting **isBuffered** to 0, **environment** to NULL)
 - Remove frame from the end of the “**free_frame_list**”.

- Bring it to the head of the "**free_frame_list**".
- c) Set its entry in the page table to **NULL** to indicate that this page is no longer exists (see [Appendix III](#)).
- 2. **Remove ONLY working set pages** that are located in the given user virtual address range. **REMEMBER** to
 - Clear the entry inside the page table to indicate that the page is no longer exists (see [Appendix III](#)).
 - Update the working sets after removing.
- 3. **Remove ONLY the EMPTY page tables** in the given range (i.e., no pages are mapped in it)
- 4. **Remove ALL pages** in the given range **from the page file** (see [Appendix I](#)).

Testing

A- How to test your project

TESTS are available inside the Project Folder

BONUS:

FIRST: User Realloc

- Attempts to resize the allocated space at given "virtual address" to "new size" bytes, possibly moving it in the user 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_with_buffering()

SECOND: Free the entire environment (exit)

- For each BUFFERED page in the User Space
 - If the buffered page is MODIFIED then
 - Get its "Frame_info" pointer (see [Appendix X](#))
 - Remove it from the "modified_frame_list"
 - Flag it as a free frame (setting isBuffered to 0, environment to NULL)
 - Add it to the head of "free_frame_list" (by calling free_frame())
 - Else
 - Just flag its "Frame_Info" as free frame (setting isBuffered to 0, environ. to NULL)
 - Bring it to the head of the "free_frame_list"
- Then, you should free:
 1. All pages in the page working set
 2. The working set itself
 3. All page tables in the entire user virtual memory
 4. The directory table
 5. All pages from page file, and the env itself and this code *is already* written for you 😊

Enjoy developing your own OS

😊 GOOD LUCK 😊

APPENDICES

APPENDIX I: Page File Helper Functions

There are some functions that help you work with the page file. They are declared and defined in “kern/file_manager.h” and “kern/file_manager.c” respectively. Following is brief description about those functions:

Pages Functions

Add a new environment page to the page file

Function declaration:

```
int pf_add_empty_env_page( struct Env* ptr_env, uint32 virtual_address, uint8
                           initializeByZero);
```

Description:

Add a new environment page with the given virtual address to the page file and initialize it by zeros.

Parameters:

ptr_env: pointer to the environment that you want to add the page for it.

virtual_address: the virtual address of the page to be added.

initializeByZero: indicate whether you want to initialize the new page by ZEROs or not.

Return value:

= 0: the page is added successfully to the page file.

= E_NO_PAGE_FILE_SPACE: the page file is full, can't add any more pages to it.

Example:

In dynamic allocation: let for example we want to dynamically allocate 1 page at the beginning of the heap (i.e. at address USER_HEAP_START) without initializing it, so we need to add this page to the page file as follows:

```
int ret = pf_add_empty_env_page(ptr_env, USER_HEAP_START, 0);

if (ret == E_NO_PAGE_FILE_SPACE)

    panic("ERROR: No enough virtual space on the page file");
```

Read an environment page from the page file to the main memory

Function declaration:

```
int pf_read_env_page(struct Env* ptr_env, void *virtual_address);
```

Description:

Read an existing environment page at the given virtual address from the page file.

Parameters:

ptr_env: pointer to the environment that you want to read its page from the page file.

virtual_address: the virtual address of the page to be read.

Return value:

= 0: the page is read successfully to the given virtual address of the given environment.

= E_PAGE_NOT_EXIST_IN_PF: the page doesn't exist on the page file (i.e. no one added it before to the page file).

Example:

In placement steps: let for example there is a page fault occur at certain virtual address, then, we want to read it from the page file and place it in the main memory at the faulted virtual address as follows:

```
int ret = pf_read_env_page(ptr_env, fault_va);

if (ret == E_PAGE_NOT_EXIST_IN_PF)

{     ...     }
```

Update certain environment page in the page file by contents from the main memory

Function declaration:

```
int pf_update_env_page(struct Env* ptr_env, void *virtual_address, struct
Frame_Info* modified_page_frame_info);
```

Description:

Updates an existing page in the page file by the given frame in memory

Parameters:

ptr_env: pointer to the environment that you want to update its page on the page file.

virtual_address: the virtual address of the page to be updated.

modified_page_frame_info: the Frame_Info* related to this page.

Return value:

= 0: the page is updated successfully on the page file.

= E_PAGE_NOT_EXIST_IN_PF: the page to be updated doesn't exist on the page file (i.e. no one add it before to the page file).

Example:

```
struct Frame_Info *ptr_frame_info = get_frame_info(...);

int ret = pf_update_env_page(environment, virtual_address, ptr_frame_info);
```

APPENDIX II: Working Set Structure & Helper Functions

Working Set Structure

As stated before, each environment has a working set that is dynamically allocated at the env_create() with a given size and holds info about the currently loaded pages in memory.

It holds two important values about each page:

1. User virtual address of the page
2. Time stamp since the page is last referenced by the program (to be used in **LRU** replacement algorithm)

The working set is defined as a pointer inside the environment structure "`struct Env`" located in "inc/environment_definitions.h". Its size is set in "`page_WS_max_size`" during the `env_create()`. "`page_WS_last_index`" will point to the next location in the WS after the last set one.

```
struct WorkingSetElement {
    uint32 virtual_address; // the virtual address of the page
    uint8 empty; // if empty = 0, the entry is valid, if empty=1, entry is empty
    uint32 time_stamp ; // time stamp since this page is last referenced
};

struct Env {
    .
    .
    .
    //page working set management
    struct WorkingSetElement* ptr_pageWorkingSet;
    unsigned int page_WS_max_size;
    // used for modified clock algorithm, the next item (page) pointer
    uint32 page_WS_last_index;
};
```

Figure 8: Definitions of the working set & its index inside `struct Env`

Working Set Functions

These functions are declared and defined in "kern/memory_manager.h" and "kern/ memory_manager.c" respectively. Following are brief description about those functions:

Get Working Set Current Size

Function declaration:

```
inline uint32 env_page_ws_get_size(struct Env *e)
```

Description:

Counts the pages loaded in main memory of a given environment

Parameters:

e: pointer to the environment that you want to count its working set size

Return value:

Number of pages loaded in main memory for environment "e", (i.e. "e" working set size)

Set Virtual Address of Page in Working Set

Function declaration:

```
inline void env_page_ws_set_entry(struct Env* e, uint32 entry_index, uint32
                                virtual_address)
```

Description:

Sets the entry number "entry_index" in "e" working set to given virtual address after **ROUNDING it DOWN** to the start of page

Parameters:

e: pointer to an environment

entry_index: the working set entry index to set the given virtual address

virtual_address: the virtual address to set (should be ROUNDED DOWN)

Clear Entry in Working Set

Function declaration:

```
inline void env_page_ws_clear_entry(struct Env* e, uint32 entry_index)
```

Description:

Clears (make empty) the entry at “entry_index” in “e” working set.

Parameters:

e: pointer to an environment

entry_index: working set entry index

Check If Working Set Entry is Empty

Function declaration:

```
inline uint32 env_page_ws_is_entry_empty(struct Env* e, uint32 entry_index)
```

Description:

Returns a value indicating whether the entry at “entry_index” in env “e” working set is empty

Parameters:

e: pointer to an environment

entry_index: working set entry index

Return value:

0: if the working set entry at “entry_index” is NOT empty

1: if the working set entry at “entry_index” is empty

Print Working Set

Function declaration:

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

Description:

Print the page working set together with the used, modified and buffered bits + time stamp. It also shows where the `last_ws_index` of the working set is point to.

Parameters:

e: pointer to an environment

APPENDIX III: Manipulating permissions in page tables and directory

Permissions in Page Table

Set Page Permission

Function declaration:

```
inline void pt_set_page_permissions(struct Env* ptr_env, uint32 virtual_address,
                                   uint32 permissions_to_set, uint32 permissions_to_clear)
```

Description:

Sets the permissions given by “**permissions_to_set**” to “1” in the page table entry of the given page (virtual address), and **Clears** the permissions given by “**permissions_to_clear**”. The environment used is the one given by “ptr_env”

Parameters:

ptr_env: pointer to environment that you should work on

virtual_address: any virtual address of the page

permissions_to_set: page permissions to be set to 1

permissions_to_clear: page permissions to be set to 0

Examples:

1. to set page PERM_WRITEABLE bit to 1 and set PERM_PRESENT to 0

```
pt_set_page_permissions(environment, virtual_address,
                        PERM_WRITEABLE, PERM_PRESENT);
```

2. to set PERM_MODIFIED to 0

```
pt_set_page_permissions(environment, virtual_address, 0,
                        PERM_MODIFIED);
```

Get Page Permission

Function declaration:

```
inline uint32 pt_get_page_permissions(struct Env* ptr_env, uint32 virtual_address )
```

Description:

Returns all permissions bits for the given page (virtual address) in the given environment page directory (ptr_pgdir)

Parameters:

ptr_env: pointer to environment that you should work on

virtual_address: any virtual address of the page

Return value:

Unsigned integer containing all permissions bits for the given page

Example:

To check if a page is modified:

```
uint32 page_permissions = pt_get_page_permissions(environment, virtual_address);
if (page_permissions & PERM_MODIFIED)
{
    . . .
}
```

Clear Page Table Entry

Function declaration:

```
inline void pt_clear_page_table_entry(struct Env* ptr_env, uint32 virtual_address)
```

Description:

Set the entry of the given page inside the page table to **NULL**. This indicates that the page is no longer exists in the memory.

Parameters:

`ptr_env`: pointer to environment that you should work on

`virtual_address`: any virtual address inside the page

Permissions in Page Directory

Clear Page Dir Entry

Function declaration:

```
inline void pd_clear_page_dir_entry(struct Env* ptr_env, uint32 virtual_address)
```

Description:

Set the entry of the page table inside the page directory to **NULL**. This indicates that the page table, which contains the given virtual address, becomes no longer exists in the whole system (memory and page file).

Parameters:

`ptr_env`: pointer to environment that you should work on

`virtual_address`: any virtual address inside the range that is covered by the page table

Check if a Table is Used

Function declaration:

```
inline uint32 pd_is_table_used(Env* ptr_environment, uint32 virtual_address)
```

Description:

Returns a value indicating whether the table at “`virtual_address`” was used by the processor

Parameters:

`ptr_environment`: pointer to environment

`virtual_address`: any virtual address inside the table

Return value:

0: if the table at “`virtual_address`” is not used (accessed) by the processor

1: if the table at “`virtual_address`” is used (accessed) by the processor

Example:

```
if(pd_is_table_used(faulted_env, virtual_address))
{
    ...
}
```

Set a Table to be Unused

Function declaration:

```
inline void pd_set_table_unused(Env* ptr_environment, uint32 virtual_address)
```

Description:

Clears the “Used Bit” of the table at `virtual_address` in the given directory

Parameters:

`ptr_environment`: pointer to environment

`virtual_address`: any virtual address inside the table

APPENDIX IV: Page Buffering

Structures

In order to keep track of the buffered frames, the **following changes were added to FOS**:

1- We use one of the available bits in the page table as a BUFFERED bit (Bit number 9) to indicate whether the **frame of this page** is buffered or not.

31																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																					
----	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

P: Present bit (**PERM_PRESENT**)
R/W: Read/Write bit (**PERM_WRITABLE**)
U/S: User/Supervisor bit (**PERM_USER**)
A: Accessed bit (**PERM_USED**)
D: Dirty bit (**PERM_MODIFIED**)
B: Buffered bit (**PERM_BUFFERED**)
AVAIL: Available for system programmers use
NOTE: 0 indicates Intel reserved. Don't define.

2- Add the following information to the "**Frame_Info**" structure:

- a- **isBuffered**: to indicate whether **this frame** is buffered or not
- b- **va**: virtual address of the page that was mapped to this buffered frame
- c- **environment**: the environment that own this virtual address

3- Create a new list called "**modified_frame_list**"

A new Frame_Info* linked list is added to FOS to keep track of buffered modified page frames,(see memory_manager.c):

```
struct Linked_List modified_frame_list;
```

Functions

Add Frame to the Tail of a Buffered List

Description:

Adds the given frame (ptr_frame_info) to the **TAIL** of the given list (bufferList)

Function declaration:

```
inline void bufferList_add_page(struct Linked_List* bufferList, struct Frame_Info  
                               *ptr_frame_info)
```

Parameters:

bufferList: pointer to a linked list of Frame_Info* elements (either F.F.L or Modified List)
ptr_frame_info: a pointer to Frame_Info object that will be added

Example:

```
struct Frame_Info* ptr_victim_frame;
. . .
bufferList_add_page(&modified_frame_list, ptr_victim_frame);
```

Remove Frame from a Buffered List

Description:

Removes the given frame (ptr_frame_info) from the given list (bufferList)

Function declaration:

```
inline void bufferlist_remove_page(struct Linked_List* bufferList, struct Frame_Info
                                   *ptr_frame_info)
```

Parameters:

bufferList: pointer to a linked list of Frame_Info* elements (either F.F.L or Modified List)

ptr_frame_info: a pointer to Frame_Info object that will be removed

Example:

```
struct Frame_Info* ptr_victim_frame;
. . .
bufferList_add_page(&modified_frame_list, ptr_victim_frame);
```

Get Current Size of a Buffered List

Description:

Return the size of the given list (bufferList)

Function declaration:

```
LIST_SIZE(struct Linked_List* bufferList)
```

Parameters:

bufferList: pointer to a linked list of Frame_Info* elements (either F.F.L or Modified List)

Example:

```
//Get size of modified frame list
uint32 size = LIST_SIZE(&modified_frame_list);
```

Get Maximum Length of the Modified Buffered List

Description:

Return the maximum length of the modified buffered list

Function declaration:

```
uint32 getModifiedBufferLength()
```

Parameters:

No parameters

Example:

```
//Get maximum length of modified buffer list
uint32 max_len = getModifiedBufferLength();
```

Iterate on ALL Elements of a Buffered List

Description:

Used to loop on all frames in the given list

Function declaration:

```
LIST_FOREACH (Frame_Info* iterator, Linked_List* list)
```

Parameters:

list: pointer to the linked list to loop on its elements (of type `Frame_Info*`)

iterator: pointer to the current element in the list (of type `Frame_Info*`)

Example:

```
//Check which modified frame belongs to the current environment
struct Frame_Info *ptr_fi ;
LIST_FOREACH(ptr_fi, &modified_frame_list)
{
    if (ptr_fi->environment == curenv)
    {
        ...
    }
}
```

APPENDIX V: Modified CLOCK Replacement Algorithm

Modified clock replacement is a slightly modified version of normal clock algorithm you already studied, instead of working only on 1 “**used bit**”, another bit is also used called “**modified bit**”

Algorithm Description

Starting with current pointer position in the working set:

Try 1: (search for a “not used, not modified” victim page)

- Search for page with used bit = 0 and modified bit = 0
 - If a page is found, it will be the victim. **Replace it** and then update the pointer to point the next page to the victim in the working set, and algorithm is finished
- If the pointer reaches its first position again without finding a victim, goto **Try 2**

Try 2: (normal clock: search for a “not used, (regardless of the value of modified bit)” victim page)

- Search for page with used bit = 0, regardless the value of modified bit, **and setting the used bit value of any page in the way to 0**
 - If a page is found, it will be the victim, **Replace it** and then update the pointer to point the next page to the victim in the working set, and algorithm is finished
- If the pointer reaches its first position again without finding a victim, goto **Try 1**

Example

Pointer	Page	used	modified
*	P1	0	1
	P2	1	0
	P3	1	0

A
New Page:
P5 (write)

Pointer	Page	used	modified
*	P1	0	1
	P2	1	0
	P3	1	0

B
P1 is victim found
in Try 2

Pointer	Page	used	modified
	P5	1	1
*	P2	1	0
	P3	1	0

C
requested P5 is
placed

Pointer	Page	used	modified
	P5	1	1
*	P2	1	1
	P3	1	0

D
P2 is modified
New Page:
P7 (read)

Pointer	Page	used	modified
	P1	0	1
	P2	0	1
*	P3	0	0

E
P3 is victim found
in second Try 1

Pointer	Page	used	modified
*	P1	0	1
	P2	0	1
	P7	1	0

F
P7 is placed

Figure 9: Example on modified clock algorithm

Appendix VI: Command Prompt

NEW Command Prompt Features

The following features are newly added to the FOS command prompt. They are originally developed by **Mohamed Raafat & Mohamed Yousry, 3rd year student, FCIS, 2017**, thanks to them. Edited and modified by TA\ Ghada Hamed.

First: DOSKEY

Allows the user to retrieve recently used commands in (FOS>_) command prompt via UP/DOWN arrows.
Moves left and right to edit the written command via LEFT/RIGHT arrows.

Second: TAB Auto-Complete

Allow the user to auto-complete the command by writing one or more initial character(s) then press "TAB" button to complete the command. If there're 2 or more commands with the same initials, then it displays them one-by-one at the same line.

The same feature is also available for auto-completing the "Program Name" after "load" and "run" commands.

Appendix VII: Basic and Helper Memory Management Functions

Basic Functions

The basic **memory manager functions** that you may need to use are defined in "kern/memory_manager.c" file:

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	Used by "map_frame" to get a pointer to the page table if exist
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

Helpers Functions

There are some **helper functions** that we may need to use them in the rest of the course:

Function	Description	Defined in...
LIST_FOREACH (Frame_Info*, Linked_List *)	Used to traverse a linked list. Example: to traverse the modified list struct Frame_Info* ptr_frame_info = NULL; LIST_FOREACH(ptr_frame_info, &modified_frame_list) {	inc/queue.h

 }	
to_frame_number (Frame_Info *)	Return the frame number of the corresponding Frame_Info element	Kern/memory_manager.h
to_physical_address (Frame_Info *)	Return the start physical address of the frame corresponding to Frame_Info element	Kern/memory_manager.h
to_frame_info (uint32 phys_addr)	Return a pointer to the Frame_Info corresponding to the given physical address	Kern/memory_manager.h

Function	Description	Defined in...
PDX (uint32 virtual address)	Gets the page directory index in the given virtual address (10 bits from 22 – 31).	Inc/mmu.h
PTX (uint32 virtual address)	Gets the page table index in the given virtual address (10 bits from 12 – 21).	Inc/mmu.h
ROUNDUP (uint32 value, uint32 align)	Rounds a given “value” to the nearest upper value that is divisible by “align”.	Inc/types.h
ROUNDDOWN (uint32 value, uint32 align)	Rounds a given “value” to the nearest lower value that is divisible by “align”.	Inc/types.h
tlb_invalidate (uint32* page_directory, uint32 virtual address)	Refresh the cache memory (TLB) to remove the given virtual address from it.	Kern/helpers.c
rcr3()	Read the physical address of the current page directory which is loaded in CR3	Inc/x86.h
lcr3(uint32 physical address of directory)	Load the given physical address of the page directory into CR3	Inc/x86.h

Have a nice & useful project

 **GOOD LUCK** 