Operating Systems

FOS Kernel Project

Original Project Contributions and Credits goes to

Faculty of Computer and Information Sciences

Ain Shams University

2024 – 2025

Contents

INTRODUCTION **5**

What's new?! **5**

Project Specifications **6**

New Concepts *(Needed in MS2)* **7**

FIRST: Working Sets 7

SECOND: Page File 7

THIRD: System Calls 8

OVERALL VIEW **8**

DETAILS **10**

FIRST: Kernel Heap 10

TESTING **13**

A- How to test your project **13**

APPENDIX I: PAGE FILE HELPER FUNCTIONS **14**

Pages Functions **14**

Add a new environment page to the page file 14

Read an environment page from the page file to the main memory 14

Update certain environment page in the page file by contents from the main memory 15

Remove an existing environment page from the page file 15

APPENDIX II: WORKING SET STRUCTURE & HELPER FUNCTIONS **17**

Working Set Structure **17**

Working Set Functions **18**

Get Working Set Current Size 18

Get Virtual Address of Page in Working Set 18

Set Virtual Address of Page in Working Set 19

Clear Entry in Working Set 19

Check If Working Set Entry is Empty 19

Print Working Set 20

Flush certain Virtual Address from Working Set 20

APPENDIX III: MANIPULATING PERMISSIONS IN PAGE TABLES AND DIRECTORY 21

Permissions in Page Table **21**

Set Page Permission 21

Get Page Permission 21

Clear Page Table Entry 22

Permissions in Page Directory **22**

Clear Page Dir Entry 22

Check if a Table is Used 22

Set a Table to be Unused 23

APPENDIX V: BASIC AND HELPER MEMORY MANAGEMENT FUNCTIONS 24

Basic Functions **24**

Helpers Functions **24**

APPENDIX VI: COMMAND PROMPT 26

Ready-Made Commands **26**

Run process 26

Load process 26

Kill process 26

Run all loaded processes 26

Print all processes 26

Kill all processes 27

Print current scheduler method (round robin, MLFQ, …) 27

Change the Scheduler to Round Robin 27

Change the Scheduler to MLFQ 27

Print current replacement policy (clock, LRU, …) 27

Changing replacement policy (clock, LRU, …) 27

Print current kernel heap placement strategy (CONT ALLOC, NEXT FIT, BEST FIT, …) 27

Changing kernel heap placement strategy (NEXT FIT, BEST FIT, …) 27

NEW Command Prompt Features **28**

First: DOSKEY 28

Second: TAB Auto-Complete 28

***LOGISTICS:* refer to powerpoint presentation in the project materials**

Group Members: *4-6*

Delivery:

1. ***Dropbox-based***
2. ***Milestone 1****: THU* ***(17 APRIL till 10:00 PM)*** *➔ kernel heap functions*
3. ***Milestone 2: (will be announced)*** *➔ user heap functions, page fault, env free.*

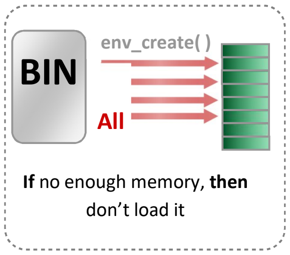
Support Dates: *WEEKLY Office Hours+*

- *It's* ***FINAL*** *delivery.*
- ***MUST*** *deliver the required tasks and* ***ENSURE*** *they're working correctly.*
- ***Code****: Make sure you are working on the* ***FOS_PROJECT_2025_template.zip*** *provided to you; Follow the shared steps to import the project folder into 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:

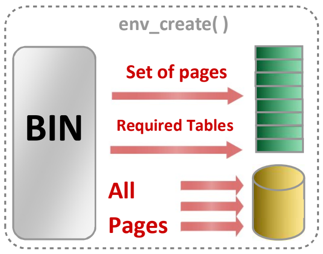


But, wait a minute…

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 project: 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 to add new features for your FOS, 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- **User heap:** allow the user to dynamically allocate and free memory space at run-time.

3- Handle **page faults** during execution**.**

4- **ENV FREE.**

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.

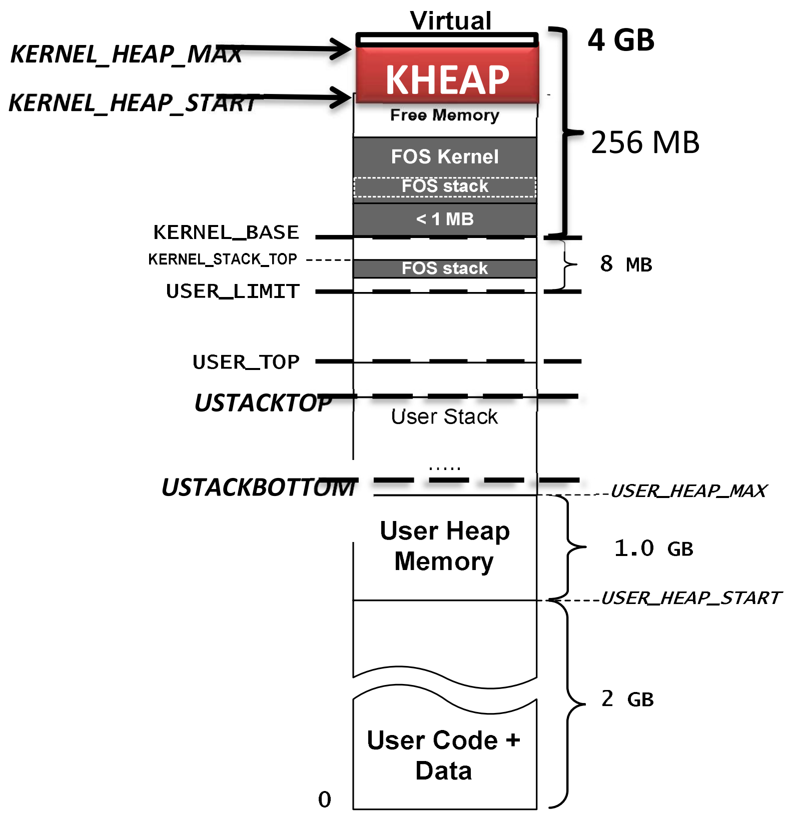


Figure 1: Virtual space layout of single user loaded program

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 *(Needed in MS2)*

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 differing from one program to another. It's 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.

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

*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.*

SECOND: Page File

The page file is an area in the secondary memory (H.D.D.) that is used for saving and loading programs 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. These functions provide the following facilities:

1- Add a new environment empty page to the page file.

2- Read an environment page from the page file to the main memory.

3- Update certain environment page in the page file from the main memory.

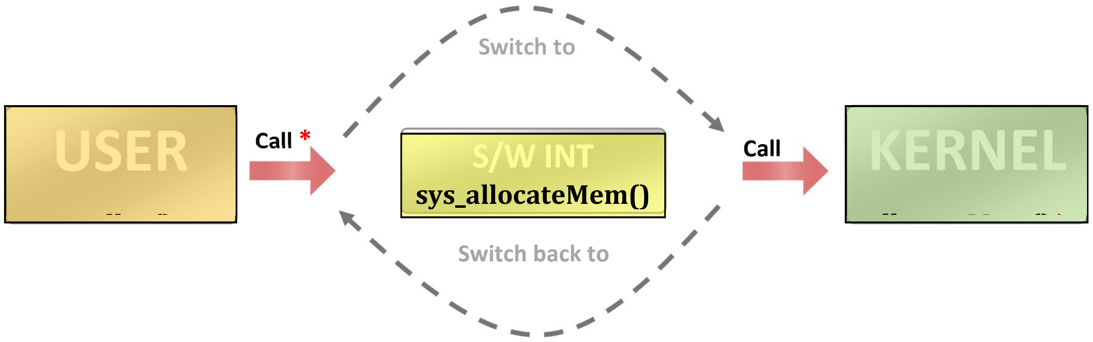
4- Remove an existing environment page from the page file.

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: 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 need 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:

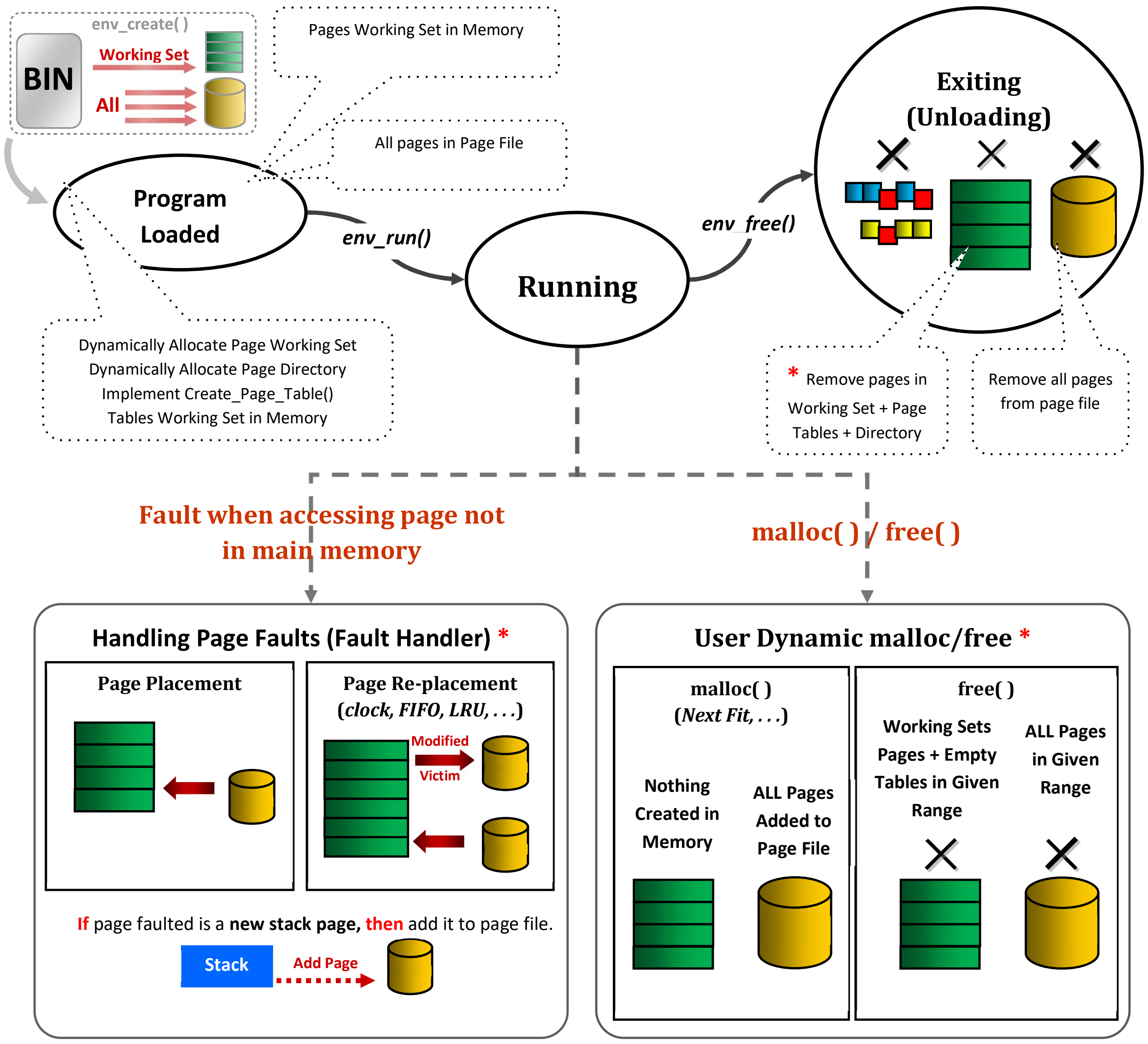


NOTE: You should do the (*) operations only

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.



env_create( )
BIN
Working Set
All
Pages Working Set in Memory
All pages in Page File
Program Loaded
Dynamically Allocate Page Working Set
Dynamically Allocate Page Directory
Implement Create_Page_Table()
Tables Working Set in Memory
env_run()
Running
env_free()
Exiting (Unloading)
* Remove pages in Working Set + Page Tables + Directory
Remove all pages from page file
Fault when accessing page not in main memory
malloc( ) / free( )
Handling Page Faults (Fault Handler) *
Page Placement
Page Re-placement (clock, FIFO, LRU, . . .)
Modified
Victim
If page faulted is a new stack page, then add it to page file.
Stack
Add Page
User Dynamic malloc/free *
malloc( ) (Next 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

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.

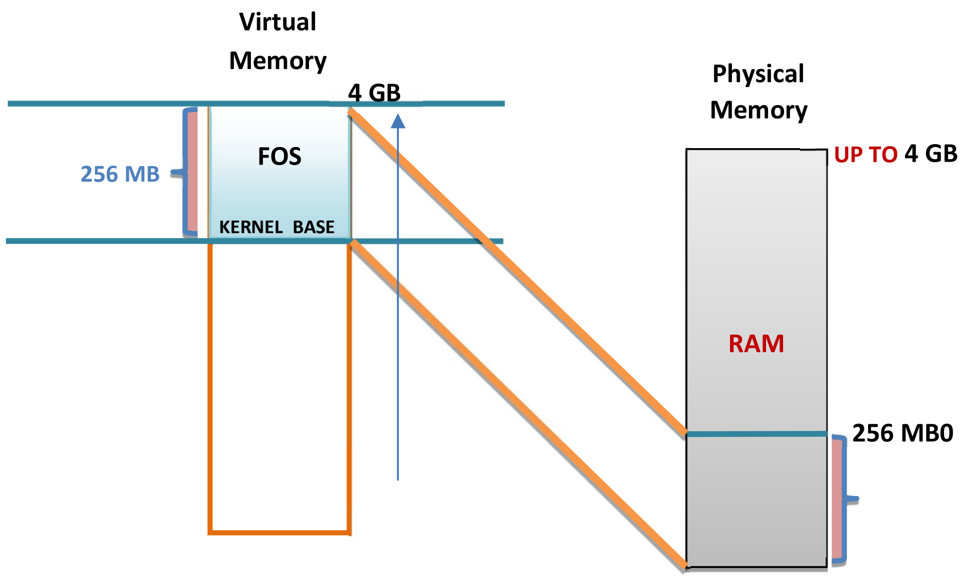


Figure 4: Mapping of the FOS VIRTUAL memory to the PHYSICAL memory [32 bit Mode]

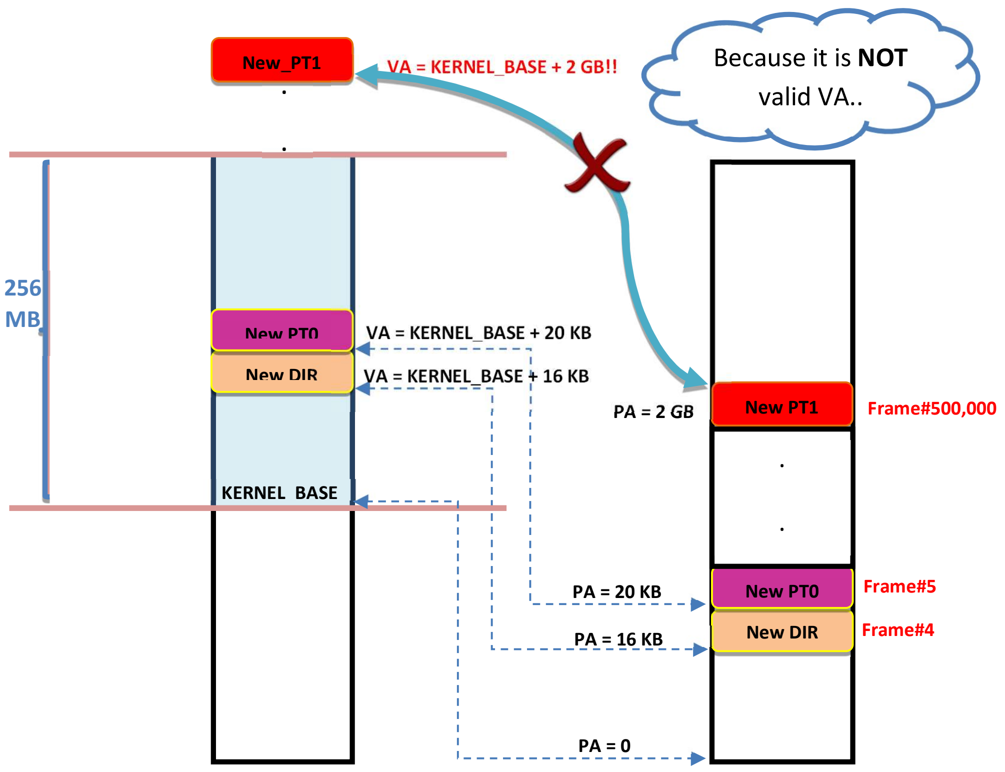


Figure 5: PROBLEM of ONE-to-ONE mappings between the FOS VIRTUAL memory and its corresponding frames in the PHYSICAL memory

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**.

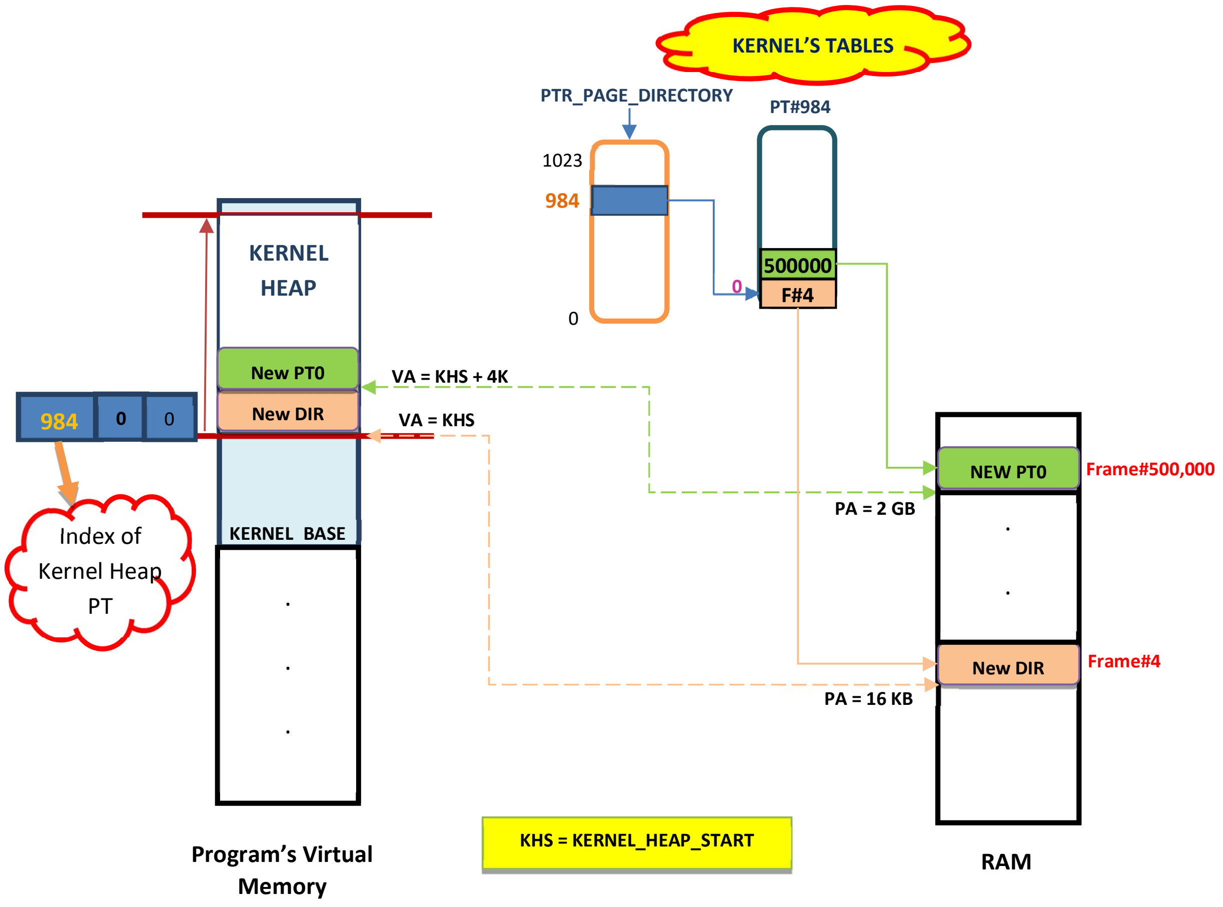


Figure 6: New Model to map kernel's VA of an allocated page in the KERNEL HEAP to pa

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 **the assigned strategy** and map it as shown in **Figure 6**. It should work as follows:

   Search for suitable place using the assigned strategy (**NEXTFIT, FIRSTFIT, BESTFIT, WORSTFIT**), if found:

   Allocate free frame(s) for the required size.

   1. 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

   Q1) 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?**

   A1) No, take care this table is one of the kernel's table that are shared with ALL loaded user programs.

   Q2) 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?

   A2) 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.

Testing

A- How to test your project

To test your implementation, a bunch of test cases programs will be used.

You can test each part from the project independently. After completing all parts, you can test the whole project using the testing scenarios described below. User programs found in "**user/**" folder.

Note: *the corresponding entries in "user_environment.c" are* ***already added for you*** *to enable FOS to run these test programs just like the other programs in "user/" folder*.

TESTS will be available on separate document isA

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);
```

Remove an existing environment page from the page file

Function declaration:

```
void pf_remove_env_page(struct Env* ptr_env, uint32 virtual_address);
```

Description:

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

Parameters:

ptr_env: pointer to the environment that you want to remove its page (or table) on the page file.

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

Example:

Let's assume for example we want to free 1 page at the beginning of the heap (i.e. at address USER_HEAP_START), so we need to remove this page from the page file as follows:

```
pf_remove_env_page(ptr_env, USER_HEAP_START);
```

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

};

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

Figure 7: 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)

Get Virtual Address of Page in Working Set

Function declaration:

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

Description:

Returns the virtual address of the page at entry “entry_index” in environment **“e”** working set

Parameters:

`e`: pointer to an environment

`entry_index`: working set entry index

Return value:

The virtual address of the page at entry “entry_index” in environment **“e”** working set

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 environment “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

Flush certain Virtual Address from Working Set

Description:

Search for the given virtual address inside the working set of **"e"** and, if found, removes its entry.

Function declaration:

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

Parameters:

`e`: pointer to an environment

`virtual_address`: the virtual address to remove from working set

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 V: 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`

Appendix VI: Command Prompt

Ready-Made Commands

Run process

Name: `run <program name> <page WS size>`

Arguments:

Program name: name of user program to load and run (should be identical to name field in UserProgramInfo array).

Page WS size: specify the max size of the page WS for this program

Description:

Load the given program into the virtual memory (RAM & Page File) then run it.

Load process

Name: `load <program name> <page WS size>`

Arguments:

Program name: name of user program to load it into the virtual memory (should be identical to name field in UserProgramInfo array).

Page WS size: specify the max size of the page WS for this program

Description:

JUST Load the given program into the virtual memory (RAM & Page File) but **don't run** it.

Kill process

Name: `kill <env ID>`

Arguments:

Env ID: ID of the environment to be killed (i.e. freeing it).

Description:

Kill the given environment by calling env_free.

Run all loaded processes

Name: `runall`

Description:

Run all programs that are previously loaded by **"ld"** command using Round Robin scheduling algorithm.

Print all processes

Name: `printall`

Description:

Print all programs' names that are currently exist in new, ready and exit queues.

Kill all processes

Name: `killall`

Description:

Kill all programs that are currently loaded in the system (new, ready and exit queues. (by calling env_free).

Print current scheduler method (round robin, MLFQ, …)

Name: `sched?`

Description:

Print the current scheduler method with its quantum(s) (RR or MLFQ).

Change the Scheduler to Round Robin

Name: **`schedRR`** `<quantum in ms>`

Description:

Change the scheduler to round robin with the given quantum (in ms).

Change the Scheduler to MLFQ

Name: **`schedMLFQ`** `<number of levels> <1st quantum> <2nd quantum>` …

Description:

Change the scheduler to MLFQ with the given number of levels and their quantums (in ms).

Print current replacement policy (clock, LRU, …)

Name: `rep?`

Description:

Print the current page replacement algorithm (CLOCK, LRU, FIFO or modifiedCLOCK).

Changing replacement policy (clock, LRU, …)

Name: `nclock` *(*`clock, lru, fifo, modifiedclock`*)*

Description:

Set the current page replacement algorithm to CLOCK (LRU, FIFO, modifiedCLOCK or Nth Chance CLOCK).

Print current kernel heap placement strategy (CONT ALLOC, NEXT FIT, BEST FIT, …)

Name: `kheap?`

Description:

Print the current KERNEL heap placement strategy (CONT ALLOC, NEXT FIT, BEST FIT, …).

Changing kernel heap placement strategy (NEXT FIT, BEST FIT, …)

Name: `khcontalloc` *(*`khnextfit, khbestfit, khfirstfit, khworstfit`*)*

Description:

Set the current KERNEL heap placement strategy to NEXT FIT (BEST FIT, …).

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.

Have a nice & useful project

☺ GOOD LUCK ☺