Faculty of Computer and Information Sciences

Ain Shams University

Third Year – First Semester

2021 - 2022

Operating Systems

FOS KERNEL PROJECT

Contents

LOGISTICS: ***refer to power point presentation in the project materials***

Delivery:

1. ***Dropbox-based***
2. *Project Delivery: SUN of Week#12* ***(26 DEC till 10:00 PM)***

Support Dates:

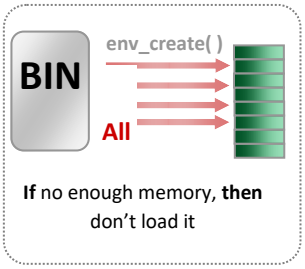
- ***Week#9 ➔ Project Explanation***
- ***Week#10 ➔ Office Hours Support***
- ***Week#11 ➔ Dedicated Support***
- ***Week#12 ➔ Project Delivery & Discussion***

– *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_2021_template.zip*** *provided to you; Follow these steps to import the project folder into the eclipse*

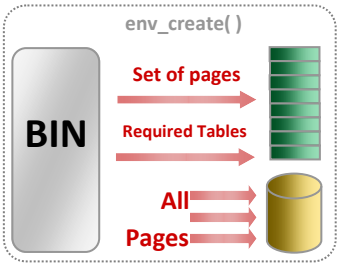
Introduction

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 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- **Load and run** multiple user programs that are **partially** loaded in main memory and **fully** loaded in secondary memory **(H.D.D.) (*DONE*)**
2- Handle **page faults** during execution by applying **LRU 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, active list and second chance lists information. Each program contains a working set which is **FIXED** size array of virtual addresses corresponding to pages that are loaded in main memory. The working set size varies from a program to the other based on its needs.

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

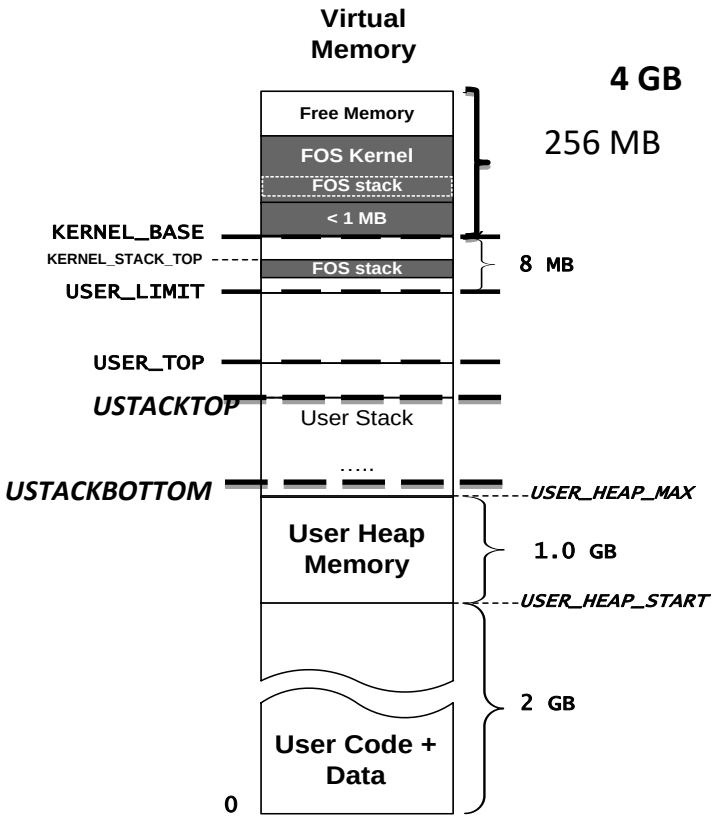


Figure 1: Virtual space layout of single user loaded program

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 and LRU Lists

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. LRU lists are **active list** and **second chance list** which are linked lists to the elements in the working lists in an order according to the APPROXIMATED LRU strategy. So, the total MAX size of both LRU lists with each other must not exceed the MAX size of the working set array.

See Appendix II for description about the required data structures, lists and helper functions.

Although it's a fixed size array, but its size is differing from one program to another. It's max size with the LRU lists are specified during the env_create() and are set inside the "struct Env", let's say ***N*** pages totally, **N/2** for the active list and **N/2** for the second chance list, 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 which is distributed on the LRU lists. The active list will contain **MAX N/2** pages and the second chance list as well which are the **N** pages in the working set array.

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 either the **ACTIVE LIST** or the **SECOND CHANCE LIST** of the program based on **LRU replacement strategy**. If both lists are complete, then one of the program environment pages from the second chance list should be removed using LRU. Then, another page is moved from the active list using FIFO to be placed at the head of the second chance list. Finally, the new faulted page will be placed at the head of the active list. This is

called **LOCAL** replacement since each program should replace one of its own loaded pages. This means that our lists are **FIXED size with LOCAL replacement facility**.

FOS must maintain the LRU lists 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 active and the second chance lists 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.

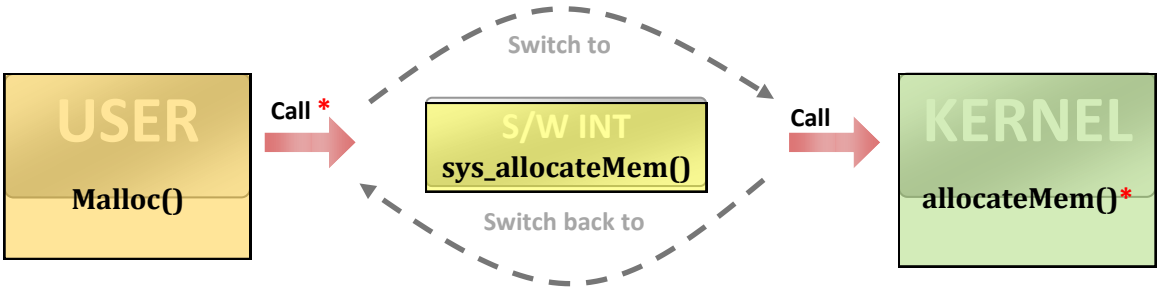
See Appendix I for description about these helper functions.

> *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

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, **t**he unmarked components are already implemented.

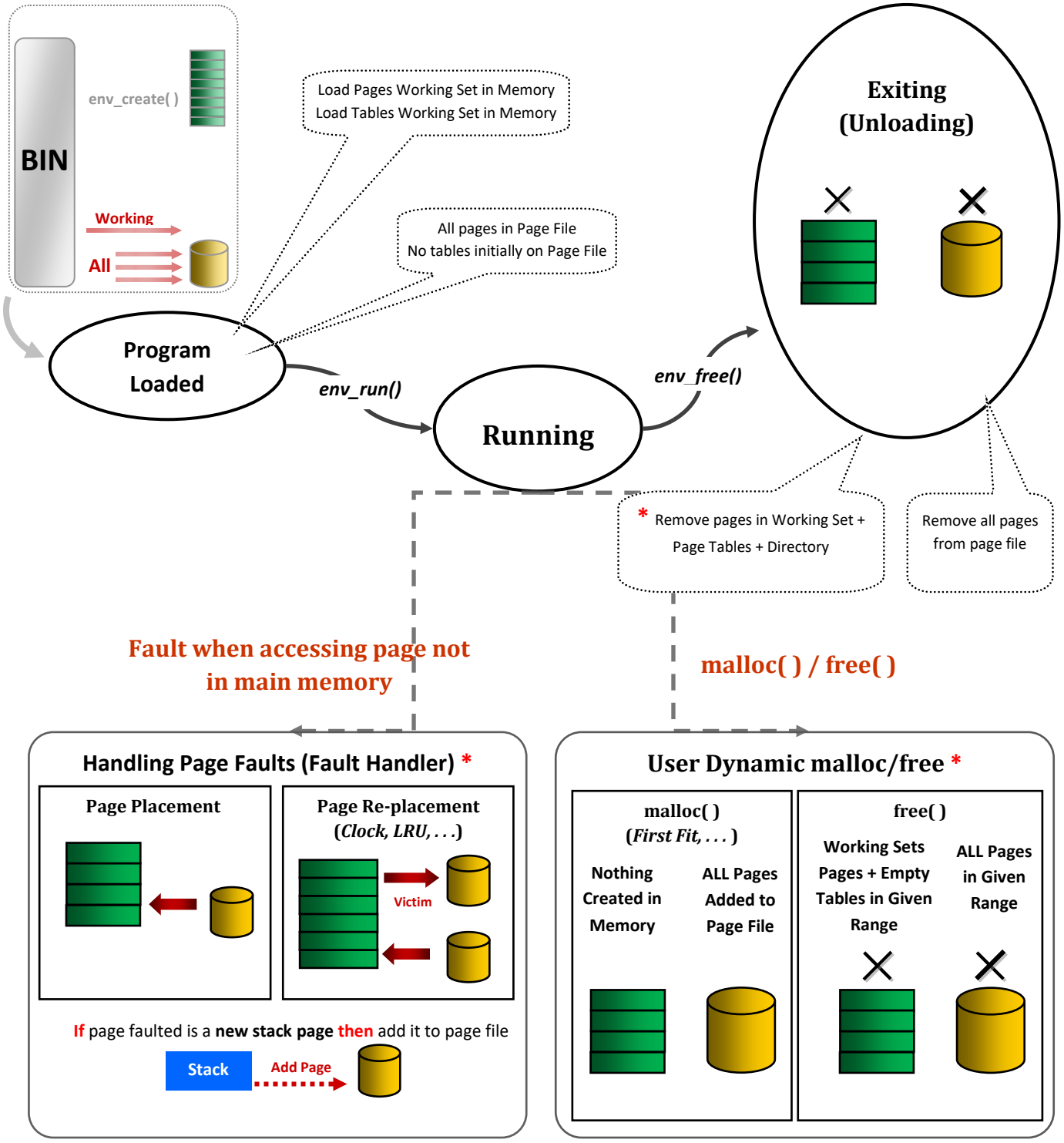


Figure 2: Interaction among components in project

Details

FIRST: Load Environment by env_create( )

in "kern/user_environment.c" (no student code needed, just understand)

After env_create() is executed to load a program binary, the binary will partially loaded in main memory (part of segments + stack page) and FULLY loaded in page file (all segments + stack page)

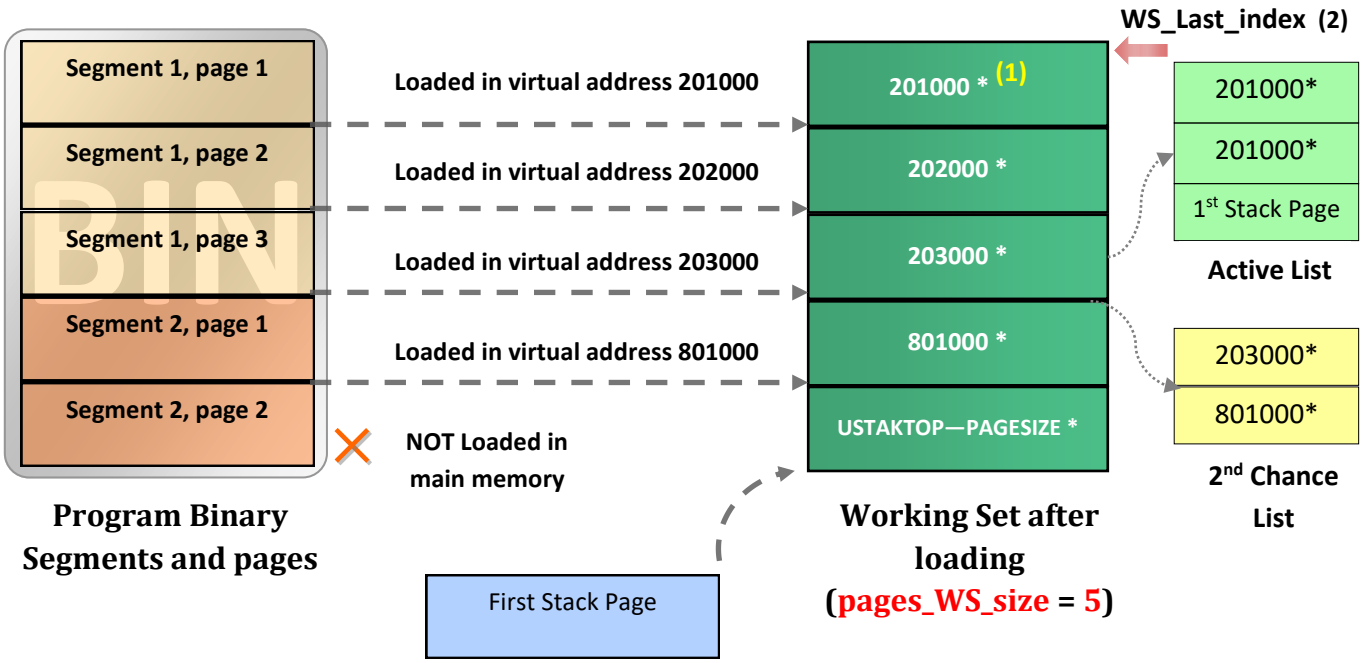


Figure 3: View of working set <u>after loading</u> a program binary with env_create(). [(1): * this asterisk means that "Used Bit" is set to 1 in page table for this virtual address, (2) "WS_Last_index": the WS last index]

SECOND: Fault Handler

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** does not exist in the main memory (i.e. new table). (See the following figure)

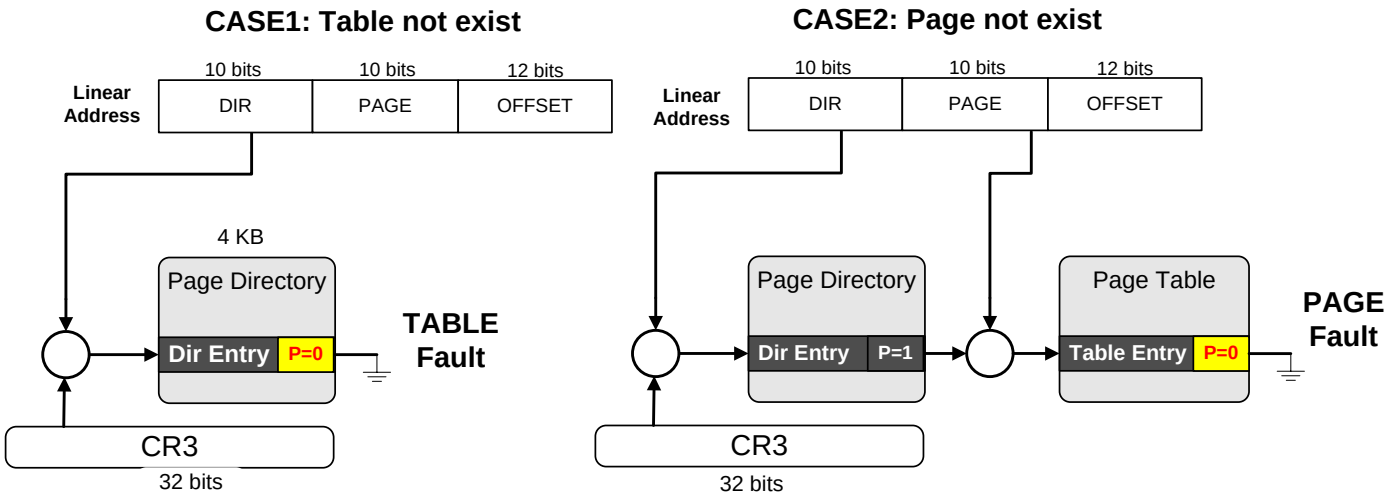


Figure 4: Fault types (table fault and page fault)

- The first case (Table Fault) is **already handled** for you by allocating a new page table if it does not exist
- You **should handle** the page fault in FOS kernel by using an **approximation for the least recently used algorithm for page replacement**. LRU chooses the victim as the page that has not been referenced for the longest time. We will implement an approximation to LRU using two lists.

IMPORTANT: Refer to the ppt inside the project materials for more details and examples

- You should implement the function "**page_fault_handler**" inside "**kern/trap.c**"

void page_fault_handler(struct Env * curenv, uint32 fault_va)

- curenv: is the environment of the current running program that requests a page not exist in MEM
- fault_va: the virtual address of a page that is required to be accessed but it doesn't exist in the main memory

- Refer to the basic fault handler structures and functions (***Appendix II***)
- Refer to the lists helper functions to be able to treat with the LRU lists (***Appendix III***)
- Refer to helper functions to deal with flags of Page Table entries (***Appendix IV***)
- Refer to the basic and helper memory manager functions (Appendix VI)

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

- if there are elements in **PageWorkingSetList** then do

Placement:

1. Check if the Active list is full, if true, remove the last element then add it to the second chance list.

2. Check if the fault_va in the Second chance list, set its PRESENT bit to 1 then add it to the Active list.

   ELSE
   1. Pull an element from the **PageWorkingSetList** for the faulted page, and add the fault_va in it.
   2. Allocate and map a frame for the faulted page.
   3. Read the faulted page from page file to memory.
   4. 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, do

Replacement:

1. Pick the LRU approximation victim from the tail of the second chance list.
2. If the victim page was modified, then:
   a. Update its page in page file (***see Appendix I***).
3. Unmap the victim frame.
4. Apply **Placement** steps as mentioned before.

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/malloc.c"**
- The "**malloc()**" , "**free**()" functions shall allocate the new size using *Best Fit Strategy.*

Allocation Using Best Fit

- You should use THIRD: System Calls to switch from user to kernel
- Your user code will be in "**malloc()**" and "**free()**" functions in "**uheap.c**"
- Your kernel code will be in "**allocateMem()**" and "**freeMem()**" functions in "**memory_manager.c**"

User Side (malloc)

1. To create new dynamic allocation, apply the best fit strategy to find a new space for the given required size.
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

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.

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)

- **Remove ONLY working set pages** that are located in the given user virtual address range. **REMEMBER** to
  - Update the working sets after removing
- **Remove ONLY the EMPTY page tables** in the given range (i.e. no pages are mapped in it)
- **Remove ALL pages** in the given range **from the page file** (see *Appendix I*).

BONUSES

FIRST: Free the entire environment (exit) by env_free()

- You should free:
  1. All pages in the page working set
  2. LRU lists
  3. All page tables in the entire user virtual memory
  4. The directory table.
  5. All pages from page file, this code ***is already*** written for you

SECOND: Freeing RAM when it's Full

When allocating new frame, if there's no free frame, then you should:

1- Remove one or more of the exited processes, if any, from the main memory (those with status ENV_EXIT) (refer to APPENDIX VI for helper functions)

2- If not, then you should free at least 1 frame from the second list of user working set of EACH process

THIRD: O(1) Implementation of Fault Handler

- Implement the main logic of re/placement in O(1) (Neglecting complexity of read/write from/to page file)
- Compare the performance of this implementation with the naïve one

FOURTH: Add "Program Priority" Feature to FOS

- Five 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!!

FIRST: Stack De-Allocation

In the env_create, when loading the program, only ONE stack page is allocated and mapped in RAM and added as a copy on the page file (H.D.D). Later, page faults can occur for stack pages that are not allocated. In this case, the page fault is handled by YOU to allocate the required stack page(s) in RAM and add it in the program's page file. These stack pages will remain in page file until closing the program (EXIT).

The problem is that this may cause LEAK in both the Page File (H.D.D) and the memory as shown in **Figure 5**. Since these allocated pages will remain on the page file until the exit (even if there're no further need for them), this can filling up the page file and no further pages can be added.

So the challenge here is to handle this occurred leak by removing the UN-NEEDED stack pages every while from both memory and its copy on the page file as well.

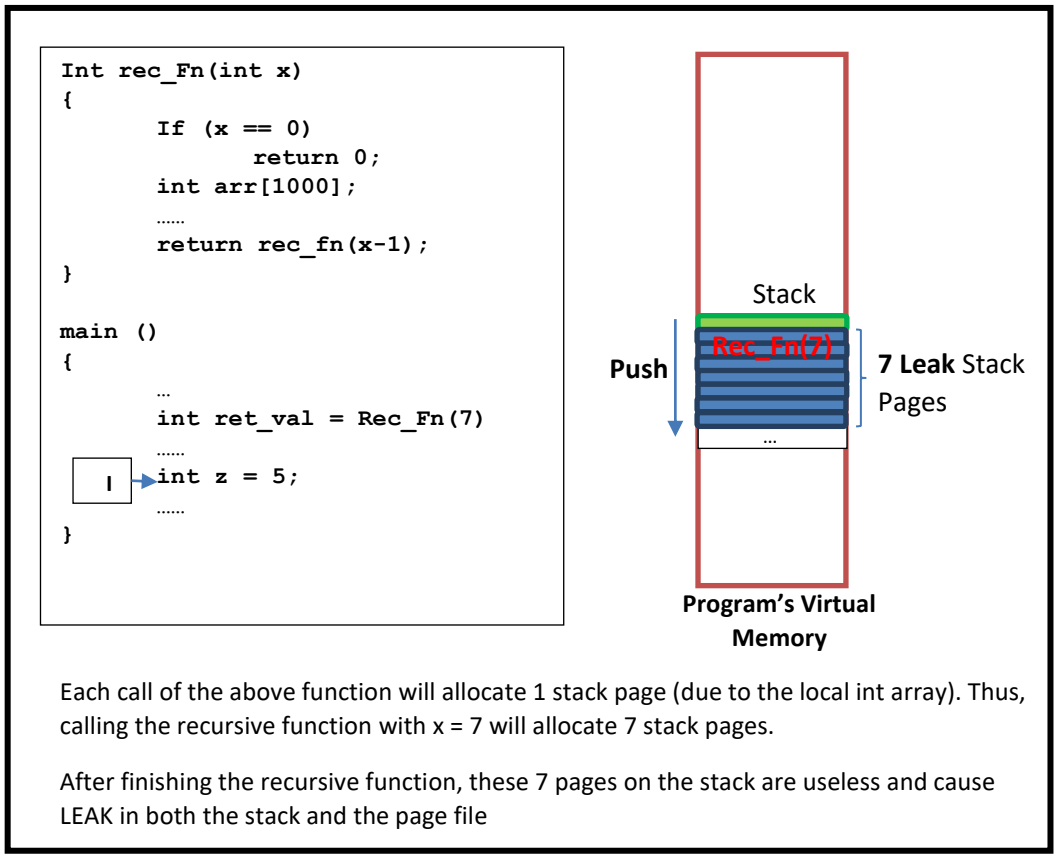


Figure 5: Memory leak due to the remaining part of the stack pages allocated in the memory during the execution of the recursive function

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 to its saved state.

THIRD: Dynamic Allocation, Local Scope Strategy

As you see, we give a fixed number of pages to each process (this is called Fixed Allocation). With this policy, it is necessary to decide ahead of time the amount of allocation to give to a process. The drawback to this approach is twofold:

1. If allocations tend to be too small, then there will be a high page fault rate, causing the entire multiprogramming system to run slowly.
2. If allocations tend to be unnecessarily large, then there will be too few programs in main memory and there will either be considerable processor idle time or considerable time spent in swapping.

To solve this problem, a **Dynamic Allocation** is used in which a process experiencing page faults will gradually grow in size, which should help reduce overall page faults in the system. But this growth is limited by Max Size that a process can't exceed (to avoid greedy process for allocating the whole memory).

The idea can be summarized as follows:

1. When a new process is loaded into main memory, allocate to it a certain number of page frames as its resident set, based on application type, program request, or other criteria.
2. When a page fault occurs, select the page to replace from among the resident set of the process that suffers the fault (Local Scope).
3. From time to time, reevaluate the allocation provided to the process, and increase or decrease it to improve overall performance.

With this strategy, the decision to increase or decrease a resident set size is a deliberate one and is based on an assessment of the likely future demands of active processes. Because of this evaluation, such a strategy may yield better performance.

The key elements of the dynamic-allocation, local-scope strategy are the criteria used to determine resident set size and the timing of changes. One specific strategy that has received much attention in the literature is known as the **working set strategy**. (For more details, refer to Textbook CH.8 starting from Page 377)

Challenge Description

Develop a dynamic-allocation, local-scope strategy that **minimizes** the number of page faults occur when running multi-programs in the FOS.

In this CHALLENGE: Every effort is welcomed, search the books, ask faculty staff, search the internet or even invent your own algorithm.

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 later 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: Page Fault Handler Structures and Helper Functions

Environment Structure

As stated before, each running instance from a specific program, a new environment is created for it using the env_create() function.

It holds the following important attributes for each program:

```
struct Env
{          …….
           struct WS_List PageWorkingSetList ;   //LRU Approx: List of available WS elements
           struct WS_List ActiveList ;           //LRU Approx: ActiveList that should work as FCFS
           struct WS_List SecondList ;           //LRU Approx: SecondList that should work as LRU
           int ActiveListSize ;                  //LRU Approx: Max allowed size of ActiveList
           int SecondListSize ;                  //LRU Approx: Max allowed size of SecondList
          …….
}
```

Figure 6: Definitions of the `struct Env`

Helper Functions

Print the content of the LRU lists

Function declaration:

```
void print_page_working_set_or_LRUlists(struct Env * env_ptr)
```

Description:

Print the content of the active list and the second chance list.

Parameters:

env_ptr: pointer to the environment that you want to display the content of its LRU lists.

APPENDIX III: Lists Helper Functions

IMPORTANT: you should pass all the lists to the functions by reference

Put & before the name of the list

Iterate on ALL Elements of a Specific List

Description:

Used to loop on all frames in the given list

Function declaration:

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

Parameters:

`list`: pointer to the linked list to loop on its elements

`iterator`: pointer to the current element in the list

Example:

```
struct WorkingSetElement *element;
LIST_FOREACH(element, &(curenv->ActiveList))
{
        //write your code.
}
```

Get the size of any list

Description:

Used to retrieve the current size of a given list

Function declaration:

```
int size = LIST_SIZE(Linked_List * list)
```

Parameters:

`list`: pointer to the linked list

Example:

```
int size = LIST_SIZE(&(curenv->ActiveList))
```

Get the last element in a list

Description:

Used to retrieve the last element in a list

Function declaration:

```
Type_inside_list* element = LIST_LAST(Linked_List * list)
```

Parameters:

`list`: pointer to the linked list

Get the first element in a list

Description:

Used to retrieve the first element in a list (what the head points to)

Function declaration:

```
Type_inside_list* element = LIST_FIRST(Linked_List * list)
```

Parameters:

`list`: pointer to the linked list

Remove a specific element in a list

Description:

Used to remove an given element from a list

Function declaration:

```
LIST_REMOVE(Linked_List * list, Type_inside_list* element)
```

Parameters:

`list`: pointer to the linked list

`element`: is the element to be removed from the given list

Insert a new element at the BEGINNING of a list

Description:

Used to insert a new element at the head of a list

Function declaration:

```
LIST_INSERT_HEAD(Linked_List * list, Type_inside_list* element)
```

Parameters:

`list`: pointer to the linked list

`element`: the new element to be inserted at the head of list

Insert a new element at the END of a list

Description:

Used to insert a new element at the tail of a list

Function declaration:

```
LIST_INSERT_TAIL(Linked_List * list, Type_inside_list* element)
```

Parameters:

`list`: pointer to the linked list

`element`: the new element to be inserted at the tail of list

APPENDIX IV: 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: Command Prompt

Ready-Made Commands

Run process

Name: **`run <prog_name> [<page_WS_size> <LRU_second_list_size>]`**

Arguments:

prog_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 ***[OPTIONAL]***

LRU_second_list_size: specify the max size of the second chance list for this program ***[OPTIONAL]***

Description:

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

Load process

Name: **`load <prog_name> [<page_WS_size> <LRU_second_list_size>]`**

Arguments:

prog_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 ***[OPTIONAL]***

LRU_second_list_size: specify the max size of the second chance list for this program ***[OPTIONAL]***

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 replacement policy (clock, LRU, …)

Name: **rep?**

Description:

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

Changing replacement policy (clock, LRU, …)

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

Description:

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

Print current user heap placement strategy (NEXT FIT, BUDDY, BEST FIT, …)

Name: **uheap?**

Description:

Print the current USER heap placement strategy (NEXT FIT, BUDDY, BEST FIT, …).

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

Name: **uhnextfit** *(***uhbestfit, uhfirstfit, uhworstfit***)*

Description:

Set the current user 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.

Appendix VI: 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 ☺