­­­­

**Faculty of Computer and Information Sciences**

**Ain Shams University**

**Third Year – First Semester**

**2022 - 2023**

**FOS KERNEL PROJECT**

**Milestone 3 Appendix**

**Operating Systems**

Contents

[APPENDIX I: Page File Helper Functions 3](#_Toc120046906)

[Pages Functions 3](#_Toc120046907)

[Add a new environment page to the page file 3](#_Toc120046908)

[Read an environment page from the page file to the main memory 3](#_Toc120046909)

[Update certain environment page in the page file by contents from the main memory 4](#_Toc120046910)

[Remove an existing environment page from the page file 4](#_Toc120046911)

[APPENDIX II: Working Set Structure & Helper Functions 5](#_Toc120046912)

[Working Set Structure 5](#_Toc120046913)

[Working Set Functions 5](#_Toc120046914)

[Get Working Set Current Size 5](#_Toc120046915)

[Get Virtual Address of Page in Working Set 5](#_Toc120046916)

[Set Virtual Address of Page in Working Set 6](#_Toc120046917)

[Clear Entry in Working Set 6](#_Toc120046918)

[Check If Working Set Entry is Empty 6](#_Toc120046919)

[Print Working Set 7](#_Toc120046920)

[Flush certain Virtual Address from Working Set 7](#_Toc120046921)

[APPENDIX III: Semaphore Data Structures & Helper Functions 8](#_Toc120046922)

[Data Structures 8](#_Toc120046923)

[Helper Functions 8](#_Toc120046924)

[Create Semaphore Array 8](#_Toc120046925)

[Allocate Semaphore Object 8](#_Toc120046926)

[Get Semaphore ID 9](#_Toc120046927)

[Free Semaphore Object 9](#_Toc120046928)

[Add environment to the Queue 9](#_Toc120046929)

[Remove environment from the Queue 10](#_Toc120046930)

[APPENDIX IV: Scheduler Helper Functions 11](#_Toc120046931)

[Helper Functions 11](#_Toc120046932)

[Insert Environment to Ready Queue 11](#_Toc120046933)

[Remove Environment from Ready Queue 11](#_Toc120046934)

[Insert Environment to the NewEnv Queue 11](#_Toc120046935)

[Remove Environment from NewEnv Queue 11](#_Toc120046936)

[Insert Environment to the Exit Queue 11](#_Toc120046937)

[Remove Environment from Exit Queue 12](#_Toc120046938)

[APPENDIX V: Shared Variables Data Structures & Functions 13](#_Toc120046939)

[Data Structure 13](#_Toc120046940)

[Helper Functions 14](#_Toc120046941)

[Create Shares Array 14](#_Toc120046942)

[Allocate Shared Object 14](#_Toc120046943)

[Get Shared Object ID 14](#_Toc120046944)

[Free Shared Object 14](#_Toc120046945)

[Store certain frame into shared object “frames storage” 15](#_Toc120046946)

[Retrieves a certain frame from shared object “frames storage” 15](#_Toc120046947)

[Removes all stored frames from shared object “frames storage” 16](#_Toc120046948)

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/disk/pagefile\_manager.h” and “kern/disk/pagefile\_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. Used during the initial loading of a process (inside env\_create)

#### 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, uint32 virtual\_address, struct FrameInfo\* modified\_page\_frame\_info));

#### Description:

* **Updates** an existing page in the page file by the given frame in memory.
* If the page **does not exist** in page file & **belongs** to either **USER HEAP** or **STACK**, it **adds** it to the page file

#### 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 FrameInfo\* related to this page.

#### Return value:

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

= E\_NO\_PAGE\_FILE\_SPACE: the page file is full, can’t add any more pages to it.

#### Example:

struct FrameInfo \*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

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. Empty Flag

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 1: Definitions of the working set & its index inside** struct Env

## Working Set Functions

These functions are declared and defined in “kern/mem/working\_set\_manager.h” and “kern/ mem/ working\_set\_manager.c” respectively. Following is 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: Semaphore Data Structures & Helper Functions

They are declared and defined in “kern/conc/semaphore\_manager.h” and “kern/ conc/semaphore\_manager.c” respectively. Following is brief description about data structures and helper functions:

## Data Structures

A struct is defined for each semaphore containing:

1. ID of the owner environment
2. Name
3. Value
4. Queue of blocked environments on it

Then, we dynamically allocate array of semaphores with size "MAX\_SEMAPHORES", See figure 3

**Figure 11: Semaphores data structures defined in "semaphore\_manager.h"**

## Helper Functions

### Create Semaphore Array

Function declaration:

**void create\_semaphores\_array(int size)**

Description:

Dynamically allocate the array of semaphore objects “**semaphores**” and initialize it by 0's. It set the empty flag = 1. [Already called for you ☺]

Parameters:

size: max number of semaphores.

### Allocate Semaphore Object

Function declaration:

**int allocate\_semaphore\_object(struct Semaphore** \*\*allocatedObject**)**

Description:

Allocates a new (empty) semaphore object from the “**semaphores**" array.

Parameters:

allocatedObject: return parameter containing a pointer to the allocated semaphore.

Return value:

= if succeed, **semaphoreObjectID** (its index in the array).

= Else, **E\_NO\_SEMAPHORE**: if the “semaphores” array is full.

### Get Semaphore ID

Function declaration:

**int get\_semaphore\_object\_ID(int ownerID, char\* name)**

Description:

Get the array index of the given semaphore by searching the semaphores array with the given ownerID and name.

Parameters:

**ownerID**: ID of the owner environment of the required semaphore.

**name**: name of the semaphore to search for it.

Return value:

= **semaphoreObjectID** index in the array: if found.

= **E\_SEMAPHORE\_NOT\_EXISTS**: if not found.

### Free Semaphore Object

Description:

deletes the object with given “semaphoreObjectID” from the “**semaphores**" array (i.e. set its empty flag to 1 and clear all other members (name, value…)).

Function declaration:

**int free\_semaphore\_object(int semaphoreObjectID)**

Parameters:

**semaphoreObjectID**: ID of the semaphore to be removed (i.e. its index in the array).

Return value:

= 0: if found.

= **E\_SEMAPHORE\_NOT\_EXISTS**: if not found.

### Add environment to the Queue

Description:

Add the given environment into the tail of the given queue.

Function declaration:

**void enqueue(struct Env\_Queue\* queue, struct Env\* env);**

Parameters:

queue: pointer (i.e. address) to the queue to insert on it.

env: pointer to the environment to be inserted.

Example: add current environment to myQueue

struct Env\_Queue myQueue ;

...

enqueue(**&**myQueue, curenv);

### Remove environment from the Queue

Description:

Get and remove the environment from the given queue.

Function declaration:

**struct Env\* dequeue(struct Env\_Queue\* queue);**

Parameters:

queue: pointer (i.e. address) to the queue.

env: pointer to the environment to be inserted.

Return value:

pointer to the environment on the head of the queue (after removing it from the queue).

Example:

struct Env\* env;

...

env = dequeue(**&**myQueue);

# APPENDIX IV: Scheduler Helper Functions

They are declared and defined in “kern/cpu/sched.h” and “kern/cpu/sched.c” respectively. Following is brief description about these functions:

## Helper Functions

### Insert Environment to Ready Queue

Function declaration:

void sched\_insert\_ready(struct Env\* env);

Description:

Insert the given environment to the tail of the ready queue, so, it'll be scheduled by the CPU.

Parameters:

env: pointer to the environment to be inserted.

### Remove Environment from Ready Queue

Function declaration:

void sched\_remove\_ready(struct Env\* env);

Description:

Remove the given environment from the ready queue, so, it'll be NOT scheduled by the CPU.

Parameters:

env: pointer to the environment to be removed.

### Insert Environment to the NewEnv Queue

Function declaration:

void sched\_insert\_new(struct Env\* env);

Description:

Insert the given environment to the tail of the new queue to indicate that it's loaded now.

Parameters:

env: pointer to the environment to be inserted.

### Remove Environment from NewEnv Queue

Function declaration:

void sched\_remove\_new(struct Env\* env);

Description:

Remove the given environment from the new queue.

Parameters:

env: pointer to the environment to be removed.

### Insert Environment to the Exit Queue

Function declaration:

void sched\_insert\_exit(struct Env\* env);

Description:

Insert the given environment to the tail of the exit queue to indicate that it's finished now.

Parameters:

env: pointer to the environment to be inserted.

### Remove Environment from Exit Queue

Function declaration:

void sched\_remove\_exit(struct Env\* env);

Description:

Remove the given environment from the exit queue.

Parameters:

env: pointer to the environment to be removed.

# APPENDIX V: Shared Variables Data Structures & Functions

They are declared and defined in “kern/mem/shared\_memory\_manager.h” and “kern/mem/ shared\_memory\_manager.c” respectively. Following is brief description about data structures and helper functions:

## Data Structure

Each shared object has a struct that contains:

1. ID of the owner environment
2. Name of the shared variable
3. Size
4. All its frames [frames storage]
5. Sharing permissions (ReadOnly or Writable)
6. Number of environments that reference on it (share it).

Then, we dynamically allocate an array of allowed shared objects with size "MAX\_SHARES".

int MAX\_SHARES ;

///Struct that holds shared objects information

struct Share

{

///ID of the owner environment

int ownerID;

///Shared object name

char name[64];

///Shared object size

int size;

///sharing permissions (0: Read-only, 1: Writeable)

uint8 isWritable;

///to store frames to be shared

uint32 \***framesStorage**;

///references, number of envs looking at this shared memory object

uint32 references;

};

///Array of all shared objects

struct Share \***shares**;

**Figure 12: Shared object data structures defined in "shared\_memory\_manager.h"**

## Helper Functions

### Create Shares Array

Function declaration:

**void create\_shares\_array(int size)**

Description:

Dynamically allocate the array of shared objects “**shares**” and initialize it by 0's. It set the empty flag = 1. [Already called for you ☺]

Parameters:

size: max number of shared objects.

### Allocate Shared Object

Function declaration:

**int allocate\_share\_object(struct Share** \*\*allocatedObject**)**

Description:

Allocates a new (empty) shared object from the “**shares**" array and dynamically creates its "**framesStorage**".

Parameters:

allocatedObject: return parameter containing a pointer to the allocated shared object.

Return value:

= if succeed, **shareObjectID** (its index in the array).

= Else, **E\_NO\_SHARE**: if the “shares” array is full.

### Get Shared Object ID

Function declaration:

**int get\_share\_object\_ID(int ownerID, char \* shareObjectName)**

Description:

Get the array index of the shared object by searching the “shares” array with the given “ownerID” and “shareObjectName”.

Parameters:

ownerID: ID of the owner environment of the required shared object

shareObjectName: name of the shared object to find

Return value:

= **shareObjectID** index in the array: if found.

= **E\_SHARED\_MEM\_NOT\_EXISTS**: if not found.

### Free Shared Object

Function declaration:

**int free\_share\_object(int semaphoreObjectID)**

Description:

deletes the object with given “shareObjectID” from the “**shares**" array (i.e. set its empty flag to 1, delete "frames\_storage” and clear all other members (name, value…)).

Parameters:

**shareObjectID**: ID of the shared object to be removed (i.e. its index in the array).

Return value:

= 0: if found.

= **E\_SHARED\_MEM\_NOT\_EXISTS**: if not found.

### Store certain frame into shared object “frames storage”

Function declaration:

void add\_frame\_to\_storage ( uint32\* frames\_storage ,

struct FrameInfo\* ptr\_frame\_info,

uint32 index)

Description:

Store a **FrameInfo\*** inside the shared object frames storage [**Share:: framesStorage**] associated with a frame index

Parameters:

frames\_storage: the frames storage of a shared object [**Share:: framesStorage**]

ptr\_frame\_info: the FrameInfo\* to store

index: the index desired to be given of the frame after adding to the storage

*Example1*: Add a first new allocated frame (index 0) to the storage of shared object number 0

*Struct FrameInfo\* ptr\_new\_frame = 0*;

*allocate\_frame(&ptr\_new\_frame);*

**add\_frame\_to\_storage**( **shares**[0].framesStorage, *ptr\_new\_frame*, 0);

*Example2*: Add a second allocated frame to the storage of shared object number 0

**add\_frame\_to\_storage**( **shares**[0].framesStorage, *ptr\_new\_frame*, 1);

*Example3*: Add a 3rd allocated frame to the storage of shared object number 1

**add\_frame\_to\_storage**( **shares**[1].framesStorage, *ptr\_new\_frame*, 2);

### Retrieves a certain frame from shared object “frames storage”

Function declaration:

struct FrameInfo\* get\_frame\_from\_storage ( uint32\* frames\_storage ,

uint32 index)

Description:

Retrieves a **FrameInfo\*** from the shared object frames storage [**Share:: framesStorage**] associated with a frame index

Parameters:

frames\_storage: the frames storage of a shared object [**Share:: framesStorage**]

index: the index of the frame desired to be retrieved from the storage

Return value:

struct FrameInfo\* : the retrieved frame at index from frames\_storage

*Example1*:

Retrieve the first frame (index 0) of shared object 0 from the storage

*struct FrameInfo\* ptr\_new\_frame =* **get\_frame\_from\_storage** ( **shares**[0].framesStorage, 0);

*Example2*:

Retrieve the 3rd frame of shared object 1 from the storage

*struct FrameInfo\* ptr\_new\_frame =* **get\_frame\_from\_storage** ( **shares**[1].framesStorage, 2);

### Removes all stored frames from shared object “frames storage”

Function declaration:

void get\_frame\_from\_storage ( uint32\* frames\_storage )

Description:

Removes all **FrameInfo\*** instances from the shared object frames storage [**Share:: framesStorage**]

Parameters:

frames\_storage: the frames storage of a shared object [**Share:: framesStorage**] to remove all its FrameInfo\*