Faculty of Computer and Information Sciences

Ain Shams University

Third Year – First Semester

2025 - 2026

Operating Systems

FOS KERNEL PROJECT

APPENDICES

Contents

APPENDICES

APPENDIX I: String Helper Functions

Location in Code

/inc/string.h

/lib/string.c

Helper Functions

Convert from string to integer

Function prototype:

```
long strtol(const char *s, char **endptr, int base)
```

Arguments:

- S: string to be converted
- Endptr: if you want to return a pointer to the last character after finishing the conversion (set it to NULL)
- Base: number system to be used for conversion (10 for decimal, 16 for hexadecimal…)

Return:

- Long containing the integer value

String split

Function prototype:

```
int strsplit(char *string, char *SPLIT_CHARS, char **argv, int
                                                * argc)
```

Arguments:

- `string`: string to be split
- `SPLIT_CHARS`: splitting characters
- `argv`: array of strings after splitting
- `argc`: number of split strings (i.e. size of the `argv` array)

Return:

- 1 if succeed, 0 otherwise

String Length

Function prototype:

```
int strlen(const char *s)
```

Arguments:

- `s`: string to get its length

Return:

- length (# of characters) of the given string

String Comparison

Function prototype:

```
int strncmp(const char *p, const char *q, uint32 n)
```

Compare the first "n" characters from string "p" with corresponding characters in string "q"

Arguments:

- `p`: first string to be compared
- `q`: second string to compare with it
- `n`: number of characters to be compared

Return:

- 0 if identical match, +ve/-ve value otherwise

Find Character in String

Function prototype:

```
char * strfind(const char *s, char c)
```

Return a pointer to the first occurrence of 'c' in 's', or a pointer to the string-ending null character if the string has no 'c'

Arguments:

- `s`: string
- `c`: character to search with it

APPENDIX II: Lists Helper Functions

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

Put & before the name of the list

Location in Code

/inc/queue.h

Initialize the List

Description:

Initialize the given list by zeroing its size, head and tail pointers

Function declaration:

```
LIST_INIT(Linked_List* list)
```

Parameters:

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

`iterator`: pointer to the current element in 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 ELEMENTDataType *element;
LIST_FOREACH(element, &(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

Get the previous element to another element in a list

Description:

Used to retrieve the previous element to another in a list

Function declaration:

```
Type_inside_list* element = LIST_PREV(Type_inside_list* element)
```

Parameters:

`element`: is the element to get its previous

Get the next element to another element in a list

Description:

Used to retrieve the next element to another in a list

Function declaration:

```
Type_inside_list* element = LIST_NEXT(Type_inside_list* element)
```

Parameters:

`element`: is the element to get its next

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

Insert a new element AFTER a specific element in a list

Description:

Used to insert a new element after a specific element in a list

Function declaration:

```
LIST_INSERT_AFTER(Linked_List * list, Type_inside_list* listElem,
Type_inside_list* elemToInsert)
```

Parameters:

`list`: pointer to the linked list

`listElem`: the element in the list to insert after it

`elemToInsert`: the new element to be inserted after the `listElem`

Insert a new element BEFORE a specific element in a list

Description:

Used to insert a new element before a specific element in a list

Function declaration:

```
LIST_INSERT_BEFORE(Linked_List * list, Type_inside_list* listElem,
Type_inside_list* elemToInsert)
```

Parameters:

`list`: pointer to the linked list

`listElem`: the element in the list to insert before it

`elemToInsert`: the new element to be inserted before the `listElem`

APPENDIX III: ENTRY MANIPULATION in TABLES and DIRECTORY

Location in Code

/kern/mem/paging_helpers.h

/kern/mem/paging_helpers.c

Permissions in Page Table

Set Page Permission

Function declaration:

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

Description:

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

Parameters:

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

`virtual_address`: any virtual address of the page

`permissions_to_set`: page permissions to be set to 1

`permissions_to_clear`: page permissions to be set to 0

Examples:

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

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

2. to set PERM_MODIFIED to 0

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

Get Page Permission

Function declaration:

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

Description:

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

Parameters:

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

`virtual_address`: any virtual address of the page

Return value:

Unsigned integer containing all permissions bits for the given page

Example:

To check if a page is modified:

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

Clear Page Table Entry

Function declaration:

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

Description:

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

Parameters:

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

`virtual_address`: any virtual address inside the page

Permissions in Page Directory

Clear Page Dir Entry

Function declaration:

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

Description:

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

Parameters:

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

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

Check if a Table is Used

Function declaration:

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

Description:

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

Parameters:

`ptr_environment`: pointer to environment

`virtual_address`: any virtual address inside the table

Return value:

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

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

Example:

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

Set a Table to be Unused

Function declaration:

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

Description:

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

Parameters:

`ptr_environment`: pointer to environment

`virtual_address`: any virtual address inside the table

APPENDIX IV: PAGE FILE HELPER FUNCTIONS

Location in Code

/kern/disk/pagefile_manager.h

/kern/disk/pagefile_manager.c

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 V: WORKING SET STRUCTURE & HELPER FUNCTIONS

Location in Code

inc/environment_definitions.h

kern/mem/working_set_manager.h

kern/mem/working_set_manager.c

Working Set Structure

Each environment has a **working set list** (**page_WS_list**) that is initialized at the env_create(). This list should hold pointers of type **struct WorkingSetElement** containing info about the currently loaded pages in RAM.

Each struct holds two important values about each page:

1. User virtual address of the page
2. Previous & Next pointers to be used by list

It is defined inside the environment structure "struct Env" located in "inc/environment_definitions.h".

Its max size is set in "**page_WS_max_size**" during the env_create().

"**page_last_WS_element**" will point to

1. the next location in the WS after the last set one If list is full.
2. Null if the list is not full.

```
struct WorkingSetElement {
	uint32 virtual_address;  // the virtual address of the page
	LIST_ENTRY(WorkingSetElement) prev_next_info;   // list link pointers

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

Figure 1: Definitions of the working set list & its size inside `struct Env`

Working Set Functions

Create Working Set Element

Function declaration:

```
inline struct WorkingSetElement* env_page_ws_list_create_element(struct Env* e, uint32
                                 virtual_address)
```

Description:

- Create new **Working Set Element** in the kernel heap (using kmalloc()) and initialize it by the given virtual address.
- If failed, it panic()

Parameters:

`e`: pointer to an environment

`virtual_address`: virtual address to be set in this element

Return:

pointer to the created element

Print Working Set

Function declaration:

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

Description:

CASE1: If LRU List Approx. Replacement

- Print the content of the **Active List** & **Second List**.

CASE2: Else, (any other replacement)

- Print the page **Working Set List** together with the used, modified and buffered bits + time stamp. It also shows where the **`page_last_WS_element`** of the working set is point to.

Parameters:

`e`: pointer to an environment

Flush certain Virtual Address from Working Set

Function declaration:

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

Description:

CASE1: If LRU List Approx. Replacement

- Search for the given virtual address inside the **Active List** & **Second List** of **"e"** and, if found:
  - **removes** its entry from the corresponding list & **update** the lists accordingly.
  - **Unmap** its page from memory.

CASE2: Else, (any other replacement)

- Search for the given virtual address inside the **Working Set List** of **"e"** and, if found:
  - **removes** its entry from the list.
  - **Unmap** its page from memory.

Parameters:

`e`: pointer to an environment

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

APPENDIX VI: SCHEDULER STRUCTURE & HELPER FUNCTIONS

Location in Code

kern/cpu/sched.h

kern/cpu/kclock.h

kern/cpu/sched_helpers.h

kern/cpu/sched_helpers.c

Data Structures

1. Number of ready queues

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

2. Array of quantums in millisecond: to be created and initialized later during the initialization

```
 uint8 *quantums ;                     // Quantum(s) in ms
```

3. ProcessQueues struct:
    a. Queue of "New" processes
    b. Array of "Ready" queues: to be created and initialized later during the initialization
    c. Queue of "Exit" processes
    d. Protection lock "qlock"

```
struct
{
    struct spinlock qlock;               //SpinLock to protect all process queues
    struct Env_Queue env_new_queue;      // queue of all new envs
    struct Env_Queue env_exit_queue;     // queue of all exited envs
    struct Env_Queue *env_ready_queues; // Ready queue(s) for the MLFQ or RR
}ProcessQueues;
```

Helper Functions

Invoke the Scheduler

Function declaration:

```
void sched();
```

Description:

Invoke the scheduler to **context switch** into the next ready queue (if any)

Get Process

Function declaration:

```
struct Env* get_cpu_proc();
```

Description:

Get the **current** running process

Set quantum of the CPU

Function declaration:

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

Description:

Set the CPU quantum by the given quantum

Parameters:

quantum in ms

Timer Ticks

Function declaration:

```
int timer_ticks();
```

Description:

Get the current number of ticks since the beginning of the run

Initialize Queue

Description:

Initialize a new queue by setting to NULL (ZERO) its head, tail and size.

Function declaration:

```
void init_queue(struct Env_Queue* queue);
```

Parameters:

`queue`: pointer (i.e. address) to the queue to be initialized.

Example: initialize a newly created queue

```
struct Env_Queue myQueue ;

init_queue(&myQueue);
```

Get Queue Size

Description:

Get the current number of elements inside the queue.

Function declaration:

```
int queue_size(struct Env_Queue* queue);
```

Parameters:

`queue`: pointer (i.e. address) to the queue to get its size.

Example:

```
struct Env_Queue myQueue ;

...

int size = queue_size(&myQueue);
```

Enqueue Environment

Description:

Add the given environment into the head 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);
```

Dequeue Environment

Description:

Get and remove the environment from the tail of the given queue.

Function declaration:

```
struct Env* dequeue(struct Env_Queue* queue);
```

Parameters:

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

Return value:

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

Example:

```
struct Env* env;

...

env = dequeue(&myQueue);
```

Remove Environment from Queue

Description:

Remove a given environment from the queue.

Function declaration:

```
void remove_from_queue(struct Env_Queue* queue, struct Env* env);
```

Parameters:

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

`env`: pointer to the environment to be removed.

Find Environment in the Queue

Description:

Search for an environment with the given ID in the given queue.

Function declaration:

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

Parameters:

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

`envID`: environment ID to search for.

Return value:

If found: pointer to the environment with the given ID.

Else: null.

Example: find environment with ID = 1024

```
struct Env* env;

env = find_env_in_queue(&myQueue, 1024);
```

Insert Environment to the NEW Queue

Function declaration:

```
void sched_insert_new(struct Env* env);
```

Description:

Enqueue the given environment to the new queue in order to indicate that it's loaded now.

Environment status becomes NEW.

Parameters:

`env`: pointer to the environment to be inserted.

Remove Environment from NEW Queue

Function declaration:

```
void sched_remove_new(struct Env* env);
```

Description:

Remove the given environment from the new queue.

Environment status becomes UNKNOWN.

Parameters:

`env`: pointer to the environment to be removed.

Insert a NEW Environment into the READY Queue according to its Priority

Function declaration:

```
void sched_insert_ready(struct Env* env);
```

Description:

Enqueue the given environment to the ready queue corresponding to its priority index, so, it'll be scheduled by the CPU.

Environment status becomes READY.

Parameters:

`env`: pointer to the environment to be inserted.

Remove Environment from the READY Queue(s)

Function declaration:

```
void sched_remove_ready(struct Env* env);
```

Description:

Search for and remove the given environment from the ready queue(s), so, it'll be NOT scheduled anymore by the CPU.

Environment status becomes UNKNOWN.

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:

Enqueue the given environment to the exit queue to indicate that it's finished now.

Environment status becomes EXIT.

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.

Environment status becomes UNKNOWN.

Parameters:

`env`: pointer to the environment to be removed.

APPENDIX VII: PROTECTION PRIMITIVES

FIRST: Spin Lock

Location in Code

KERNEL: kern/conc/kspinlock.h, kern/conc/kspinlock.c

USER: inc/uspinlock.h, lib/uspinlock.c

Data Structures

1. **KERNEL** Spin Lock

```
struct kspinlock {
  uint32 locked;       // Is the lock held?
```

2. **USER** Spin Lock

```
struct uspinlock {
  uint32 locked;       // Is the lock held?
```

Helper Functions

Initialize the Lock

Function declaration:

```
void init_kspinlock(struct kspinlock *lk, char *name)
void init_uspinlock(struct uspinlock *lk, char *name)
```

Description:

- Initialize the KERNEL (or USER) spin lock by
  - Setting its value to 0
  - Setting its name

Acquire Kernel Lock

Function declaration:

```
void acquire_kspinlock(struct kspinlock *lk)
```

Description:

- ***Acquire the lock.*** Loops (spins) until the lock is acquired.
- Holding a lock for a long time may cause. other CPUs to waste time spinning to acquire it.
- It **disables the interrupt** to avoid deadlock (in case if interrupted from a higher priority (or event handler) just after holding the lock => the handler will stuck in busy-waiting and prevent the other from resuming)

Release Kernel Lock

Function declaration:

```
void release_kspinlock(struct kspinlock *lk)
```

Description:

- ***Release the lock.*** Setting the lock value to 0 (indicating it's free)
- It **reenable the interrupt**.

Acquire User Lock

Function declaration:

```
void acquire_uspinlock(struct uspinlock *lk)
```

Description:

- ***Acquire the lock.*** Loops (spins) until the lock is acquired.
- Holding a lock for a long time may cause. other CPUs to waste time spinning to acquire it.

Release User Lock

Function declaration:

```
void release_uspinlock(struct uspinlock *lk)
```

Description:

- ***Release the lock.*** Setting the lock value to 0 (indicating it's free)

SECOND: Sleep Lock

Location in Code

`kern/conc/sleeplock.h, kern/conc/sleeplock.c`

Data Structures

SleepLock structure containing:

1. It's value
2. Spin Lock to protect its value
3. Channel to hold the **BLOCKED** processes on this lock
4. Lock name
5. ID of the process that holds the lock

```
struct sleeplock
{
    bool locked;            // Is the lock held?
    struct spinlock lk;     // spinlock protecting this sleep lock
    struct Channel chan;    // channel to hold all blocked processes on this lock
    // For debugging:
    char name[NAMELEN];     // Name of lock.
    int pid;                // Process holding lock
};
```

Channel structure containing:

1. Queue to hold the **BLOCKED** processes on this channel
2. Channel name

```
struct Channel
{
    struct Env_Queue queue; //queue of blocked processes waiting on this channel
    char name[NAMELEN];     //channel name
};
```

Helper Functions

Initialize the Lock

Function declaration:

```
void init_sleeplock(struct sleeplock *lk, char *name)
```

Description:

- Initialize the sleep lock by
    - Setting its value to 0

- Initialize its spin lock & channel
- Setting its name

Check the Lock

Function declaration:

```
int holding_sleeplock(struct sleeplock *lk)
```

Description:

- Check whether the lock is held by the current running process or not

Invoke the Scheduler

Function declaration:

```
void sched();
```

Description:

Invoke the scheduler to **context switch** into the next ready queue (if any)

Get Process

Function declaration:

```
struct Env* get_cpu_proc();
```

Description:

Get the **current** running process

THIRD: Semaphore

Location in Code

kern/conc/ksemaphore.h, kern/conc/ksemaphore.c

Data Structures

Semaphore structure containing:

1. It's count
2. Spin Lock to protect its value
3. Channel to hold the **BLOCKED** processes on this semaphore
4. Semaphore name

```
struct ksemaphore
{
    int count;                  // Semaphore value
    struct kspinlock lk;        // spinlock protecting this count
    struct Channel chan;        // channel to hold all blocked processes on this sema
    // For debugging:
    char name[NAMELEN];         // Name of semaphore.
};
```

Channel structure containing:

1. Queue to hold the **BLOCKED** processes on this channel
2. Channel name

```
struct Channel
{
    struct Env_Queue queue; //queue of blocked processes waiting on this channel
    char name[NAMELEN];     //channel name
};
```

Helper Functions

Initialize the Semaphore

Function declaration:

```
void init_ksemaphore(struct ksemaphore *ksem, int value, char *name)
```

Description:

- Initialize the semaphore by
  - Setting its count to the given value
  - Initialize its spin lock & channel
  - Setting its name

APPENDIX VIII: MEMORY MANAGEMENT FUNCTIONS

Basic Functions

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

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 (**it keeps the AVAILABLE bits**)
`get_page_table`	Get a pointer to the page table if exist
`create_page_table`	Create a new page table by allocating a new page at the kernel heap, zeroing it and finally linking it with the directory
`unmap_frame`	Used to un-map a frame at the given virtual address, simply by clearing the page table entry (**except the AVAILABLE bits**)
`get_frame_info`	Used to get both the page table and the frame of the given virtual address
`get_page(void* va);`	**Get a page** from the Kernel Page Allocator (i.e. Allocate it) (either for **USER** or **KERNEL**). **In KERNEL**: it calls **alloc_page()** in paging_helpers.c. **In USER**: it calls **__sys_allocate_page**() which, in turn, calls also **alloc_page()**
`return_page(void* va)`	**Return a page** to the Kernel Page Allocator (i.e. Free It) (either for **USER** or **KERNEL**) **s**

Other Helpers Functions

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

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`	Refresh the cache memory (TLB) to remove the given virtual address from it.	`Kern/mem/ memory_manager.c`

address)		
tlbflush()	Clear the entire content of the TLB	inc/x86.h
get_kheap_strategy() **set_kheap_strategy** (uint32 strategy)	Get or Set the current strategy of the Kernel Heap. *KHP_PLACE_FIRSTFIT, KHP_PLACE_NEXTFIT, ...*	Kern/mem/kheap.h
get_uheap_strategy() **set_uheap_strategy** (uint32 strategy)	Get or Set the current strategy of the User Heap. *UHP_PLACE_FIRSTFIT, UHP_PLACE_NEXTFIT, ...*	Kern/mem/memory_manager.h

APPENDIX IX: SHARED VARIABLES DATA STRUCTURES & FUNCTIONS

Location in Code

kern/cmd/shared_memory_manager.h

kern/cmd/shared_memory_manager.c

Data Structure

Each shared object has a struct that contains:

1. Unique ID for this Share object
2. Proc ID of the owner environment
3. Name of the shared variable
4. Size
5. All its frames [frames storage]
6. Sharing permissions (ReadOnly or Writable)
7. Number of environments that reference on it (share it).
8. Prev-Next pointers for linked list

There's a **global Linked List** to hold all created shared objects in the kernel.

Since this list is shared among all kernel threads, it has a **spin lock** to protect it. The list and its lock are encapsulated in a single struct called **AllShares**

```
///Struct that holds shared objects information
struct Share
{
	//Unique ID for this Share object
	//Should be set to VA of created object after masking msb (to make it +ve)
	int32 ID ;
	///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;
	///list link pointers
	LIST_ENTRY(Share) prev_next_info;
};
//List of all shared objects
LIST_HEAD(Share_List, Share);
struct
{
	//List of all share variables created by any process
	struct Share_List shares_list ;
	//Use it to protect the shares_list in the kernel
	struct spinlock  shareslock;
} AllShares;
```

Figure 2: Shared object data structures defined in "shared_memory_manager.h"

Helper Functions

Initialize Sharing

Function declaration:

```
void sharing_init()
```

Description:

Initialize the shares list & its lock

Find Shared Object

Function declaration:

```
struct Share* find_share(int32 ownerID, char* name)
```

Description:

Search for shared object with the given "**ownerID**" & "**name**" in the "**shares_list**"

Parameters:

ownerID: process ID of the owner environment.

name: shared object name.

Return value:

If found: return pointer to the Share object,

else: return NULL

Get Size of Shared Object

Function declaration:

```
int size_of_shared_object(int32 ownerID, char* name)
```

Description:

Get the size of the shared object.

Can be called from the USER side using **sys_size_of_shared_object()**

Parameters:

ownerID: process ID of the owner environment.

name: shared object name.

Return value:

If found: return the size of the Share object,

else: return E_SHARED_MEM_NOT_EXISTS

APPENDIX X: COMMAND PROMPT

Location in Code

`kern/cmd/commands.h`

`kern/cmd/commands.c`

Clear Screen

Name: **`cls`**

Description:

Clear the entire screen and start accepting commands from the top left.

CREDITS:

Implemented by for Abd-Alrahman Zedan From Team Frozen-Bytes - FCIS'24-25... Thank you 😊

Run Process (for LRU Lists or Others)

Name: **`run <prog_name> <page_WS_size> [<LRU_2nd_list_size>] [<priority>]`**

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

LRU_2nd_list_size: specify the max size of the **Second Chance List** for this program ***[OPTIONAL]***

priority: specify the process priority for the **Scheduler (default 0)** ***[OPTIONAL]***

Description:

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

Load Process (for LRU Lists or Others)

Name: **`load <prog_name> <page_WS_size> [<LRU_2nd_list_size>] [<priority>]`**

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

LRU_2nd_list_size: specify the max size of the **Second Chance List** for this program ***[OPTIONAL]***

priority: specify the process priority for the **Scheduler (default 0)** ***[OPTIONAL]***

Description:

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

Set Process Priority

Name: **`setPri <envID> <priority>`**

Arguments:

envID: ID of the created environment (will be printed when execute load/run command).

priority: specify the process priority for the **Scheduler**

Description:

Set the priority of the given environment.

Set Starvation Threshold

Name: **setStarvThr <*starvationThreshold*>**

Arguments:

starvationThreshold: new value of the starvation threshold (i.e. limit on # ticks before promoting the process priority.

Description:

Set the starvation threshold of priority-based scheduler.

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

Name: **sched?**

Description:

Print the current scheduler method with its quantum(s) (RR, BSD,…).

Change the Scheduler to Priority RR

Name: **schedPRIRR** <#priorities> <quantum> <starvThresh>

Description:

Change the scheduler to Priority RR with the given # priorities, quantum (in ms) and starvation threshold

Print Current Replacement Policy (fifo, LRU, …)

Name: **rep?**

Description:

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

Change Replacement Policy (fifo, LRU, …)

Name: **nclock N** *(***fifo, clock, lru, modclock**…*)*

Description:

Set the current page replacement algorithm to CLOCK (LRU list approx, FIFO,…).

Print Current User Heap Strategy (NEXT FIT, BUDDY, BEST FIT, …)

Name: **uheap?**

Description:

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

Change User Heap Strategy (NEXT FIT, BEST FIT, …)

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

Description:

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

Print Current Kernel Heap Strategy (NEXT FIT, BEST FIT, …)

Name: **kheap?**

Description:

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

Change Kernel Heap Placement Strategy (NEXT FIT, BEST FIT, …)

Name: **khcustomfit (khbestfit***,* **khfirstfit***)*

Description:

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

APPENDIX XI: FIXED POINT OPERATIONS

Location in Code

`inc/fixed_point.h`

Functions

- Let **x** and **y** be **fixed-point p.q** numbers, let **n** be an **integer,** and **f** is **1 << q**

Convert n to fixed point:	n × f	`fix_int(int n)`
Convert x to integer (rounding toward zero):	x / f	`fix_trunc(fixed_point_t x)`
Convert x to integer (rounding to nearest):	(x + f/2) / f if x >= 0, (x - f/2) / f if x <= 0.	`fix_round(fixed_point_t x)`
Add x and y:	x + y	`fix_add(fixed_point_t x, fixed_point_t y)`
Subtract y from x:	x – y	`fix_sub(fixed_point_t x, fixed_point_t y)`
Add x and n:	x + n × f	
Subtract n from x:	x - n × f	
Multiply x by y:	((int64) x) × y / f	`fix_mul(fixed_point_t x, fixed_point_t y)`
Multiply x by n:	x × n	`fix_scale(fixed_point_t x, int n)`
Divide x by y:	((int64) x) × f / y	`fix_mul(fixed_point_t x, fixed_point_t y)`
Divide x by n:	x / n	`fix_unscale(fixed_point_t x, int n)`