Ver_B: The Buffering Effect on the Page Replacement under MLFQ Scheduler

Contents

NOTE: MAKE SURE that you followed the instructions carefully. Good Luck isA ☺

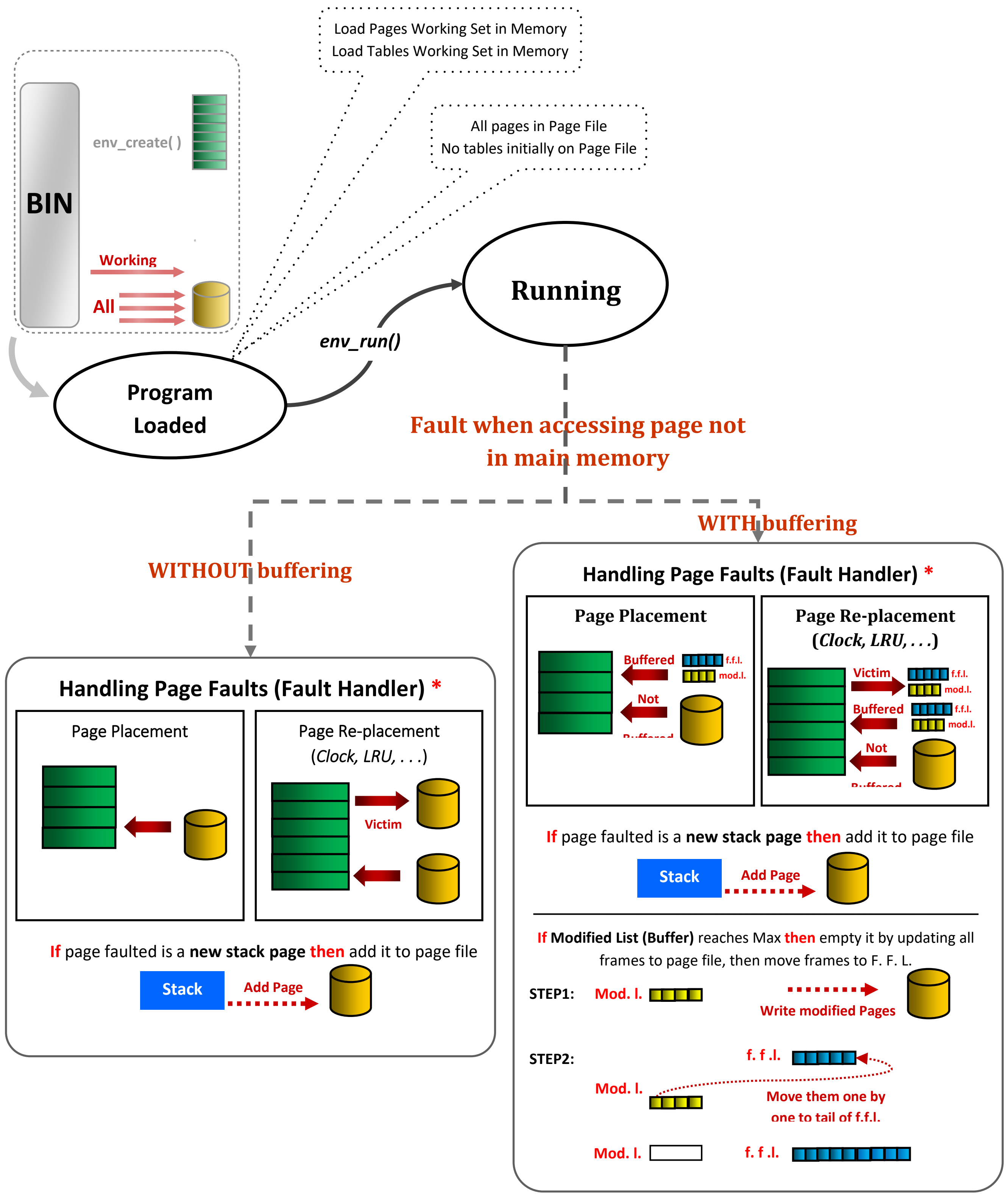
Important Instructions

Please read instructions carefully, any mistake or error may result in assignment rejection from the automated grading system:

1. To start, import the template into eclipse by following these **steps**
2. **DON'T** change any file except **kern/trap.c, kern/trap.h, kern/sched.c** and **kern/sched.h**
3. In delivery, **DON'T** rename any file
4. **PLAGIARISM** checking is applied and **strictly considered** in the evaluation

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.



Load Pages Working Set in Memory
Load Tables Working Set in Memory
All pages in Page File
No tables initially on Page File
BIN
env_create( )
Working
All
Program Loaded
env_run()
Running
Fault when accessing page not in main memory
WITHOUT buffering
WITH buffering
Handling Page Faults (Fault Handler) *
Page Placement
Page Re-placement (Clock, LRU, . . .)
Victim
If page faulted is a new stack page then add it to page file
Stack
Add Page
Buffered
Not
f.f.l.
mod.l.
If Modified List (Buffer) reaches Max then empty it by updating all frames to page file, then move frames to F. F. L.
STEP1:
Mod. l.
Write modified Pages
STEP2:
f. f .l.
Move them one by one to tail of f.f.l.

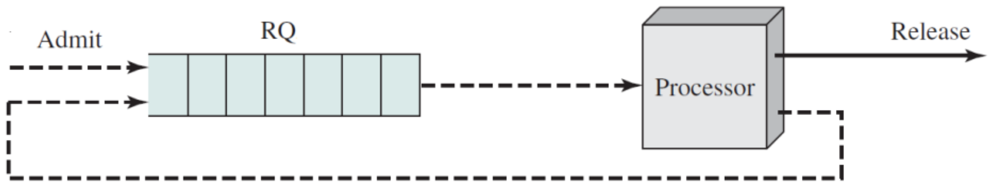
Figure 1: Interaction among components in project

Details

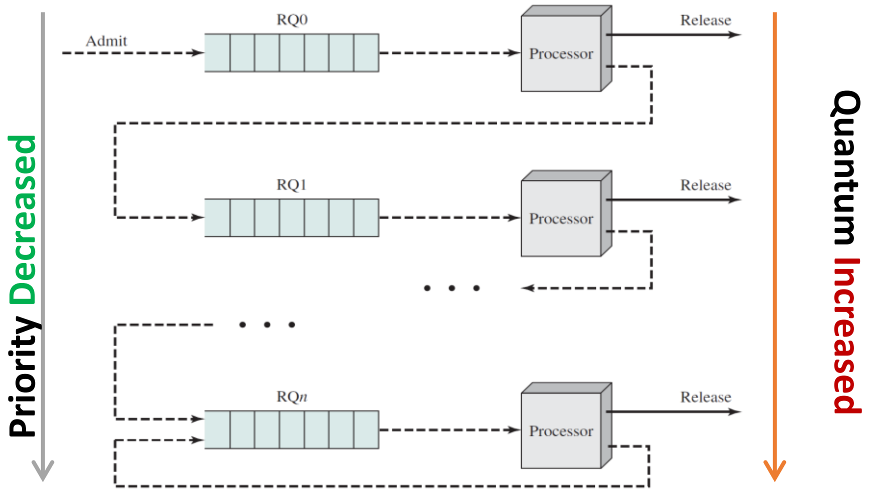
FIRST: CPU Scheduling by MLFQ

Description

- The **default** scheduler in the FOS is round robin



- Main **drawback**: favor processor-bound processes over I/O-bound processes, which results
  1. in poor performance for I/O-bound processes,
  2. inefficient use of I/O devices,
  3. an increase in the variance of response time.
- You are **asked to** implement another scheduler algorithm which is the multilevel feedback queue MLFQ.
- The main aim of the MLFQ is to establish a preference for shorter jobs by penalizing jobs that have been running longer.
- Scheduling is done on a preemptive (at time quantum) basis, and a dynamic priority mechanism is used.
- When a process first enters the system, it is placed in RQ0 (refer to Figure).
  - After its first preemption, when it returns to the Ready state, it is placed in RQ1.
  - Each subsequent time that it is preempted, it is demoted to the next lower-priority queue.
- A short process will complete quickly, without migrating very far down the hierarchy of ready queues.
  - A longer process will gradually drift downward.
  - Thus, newer, shorter processes are favored over older, longer processes.
- Within each queue, except the lowest-priority queue, a simple **FIFO** mechanism is used.
  - Once in the lowest-priority queue, a process cannot go lower, but is returned to this queue repeatedly until it completes execution.
  - Thus, this queue is treated in **round-robin** fashion.



Required Tasks:

1. CODE

- You should implement the following TWO functions "**sched_init_MLFQ**" & "**fos_scheduler**" in the KERNEL **"kern/sched.c"**

Function #1: Initialize the MLFQ (sched_init_MLFQ)

1. Create and initialize the data structures of the MLFQ:
    1. **num_of_ready_queues**
    2. Array of ready queues "**env_ready_queues**"
    3. Array of quantums "**quantums**"
2. Set the CPU quantum by the first level one

Function #2: Handle the Scheduler (fos_scheduler_MLFQ)

1. Check the existence of the **current environment** and place it in the **suitable queue**
2. **Search the queues** according to their priorities (first is highest)
3. If environment is found:
    1. Set the **CPU clock** by the quantum of the selected level
    2. Remove the selected env from its queue and return it

Helper Structures & Functions

- Refer to CPU scheduler data structures and helper functions (***Appendix III***)
- Refer to lists helper functions (***Appendix IV***)

SEEN Test Cases

IMPROTANT NOTES for TESTING:

1. There are two ways to test your code, **SEEN tests** which you have here and **UNSEEN** tests that will be used to EVALUATE you.
2. Make sure that your logic matches the specified steps exactly, since these tests do not ensure 100% that this part is totally correct.
3. Run each test in NEW SEPARATE RUN.
4. **EACH** test should run in **max of 1 min.**

tst_CPU_MLFQ_master_1.c (tmlfq1): tests the MLFQ method for CPU scheduling. It tests the placement of the current environment (if exist) in its correct queue. In addition, it tests the correct selection of the next process and setting the CPU clock by the correct quantum.

```
▪ FOS> schedMLFQ  5 2 4 6 8 10
▪ FOS> run tmlfq1 100
```

SECOND: Fault Handler

Description

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

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

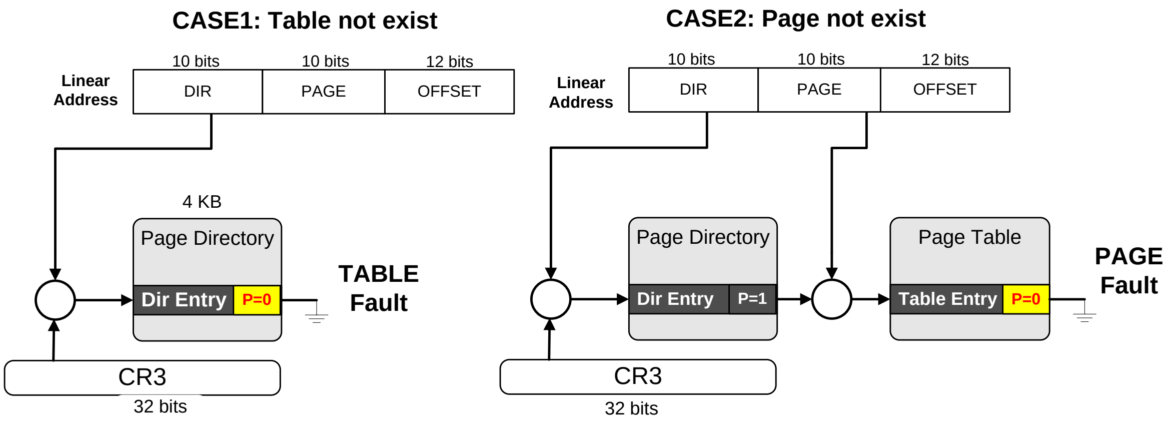


Figure 2: Fault types (table fault and page fault)

- The first case (Table Fault) is **already handled** for you by allocating a new page table if it not exists
- You **should handle** the page fault in FOS kernel by:
  - Allocate a new page for this faulted page in the main memory
  - **Loading the faulted page back to main memory from page file if the page exists. Otherwise if it is a stack page (i.e. new page), add it to the page file (for new STACK pages only). If the faulted page is not a stack page, then the fault handling shall panic, since this means that the faulted address is wrong address.**
- You can handle the page fault in **function** "**page_fault_handler()**" in "trap.c".
- In **replacement** part, it's required to apply:
  1. **MODIFIED CLOCK with buffering, check Appendix V**
  2. **MODIFIED CLOCK without buffering**

Required Tasks:

1. CODE

- You should implement both functions “**page_fault_handler**” & “**page_fault_handler_with_buffering**” inside “**kern/trap.c**”
  - **page_fault_handler:** shall handle the page fault using modified clock but **WITHOUT** buffering.
  - **page_fault_handler_with_buffering:** shall handle the page fault using modified clock but **WITH** buffering.
- The parameters of both functions are:
  - curenv: is the env of the current running program that requests a page not exist in MEM
  - fault_va: the VA of a page that is required to be accessed but it doesn’t exist in the memory

2. DOCUMENT

- After your implement your task and validate its correctness using test cases, you should prepare a document titled with the “Task name” and contains the following content:
  - Methods:
    - **Describe** briefly the modified clock and the buffering idea
    - Give two trace **examples** one for modified clock without buffering and the **same** example but with applying buffering [**different from these taken in the lecture**]
  - Results
    - As **detailed** in the last test case in the test cases section, run quick sort program for modified clock with & without buffering multiple times using the following working set sizes **[5, 50, 100, 150, 200, 250]** and for each run record the following:
      1. Time taken by each strategy (Using your stopwatch) **using ONLY WS = 5**
      2. Number of faults occurred by each strategy [Displayed at the end of running the test if it passed, only record it]
      3. Number of disk access taken by each strategy [Displayed at the end of running the test if it passed, only record it].
    - Draw 3 graph for:
      1. Time [X: Strategy, Y: Time(ms)] like **Figure 3**
      2. # of faults [X: WS Size, Y: # of faults] like **Figure 4**
      3. # of disk access [X: WS Size, Y: # of disk access] like **Figure 4**

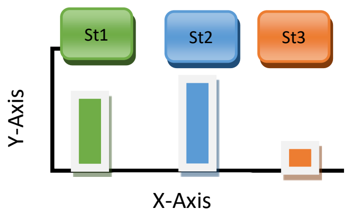


Figure 4 Graph Shape 1

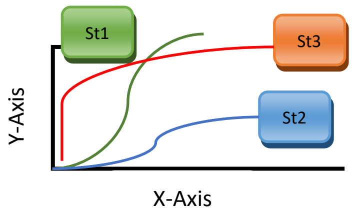


Figure 3 Graph Shape 2

  - Discussion
    - Based on your understanding and the results, discuss briefly the obtained results from <u>**EACH**</u> **graph** [For example mention which is best, which is worst and why?].
    - Comment on the number of disk reads in the buffering case, whether it’s fixed at certain number or changed over WS sizes? Why?
    - Any valuable suggestions will be considered as a **BONUS**.

Helper Structures & Functions

- Refer to helper functions to deal with the page file (***Appendix I***)
- Refer to the working set structures and helper functions (***Appendix II***)
- Refer to lists helper functions (***Appendix IV***)
- Refer to page buffering structures and function (***Appendix V***)

SEEN Test Cases

IMPROTANT NOTES for TESTING:

1. There are two ways to test your code, **SEEN tests** which you have here and **UNSEEN** tests that will be used to EVALUATE you.
2. Make sure that your logic matches the specified steps exactly, since these tests do not ensure 100% that this part is totally correct.
3. Run each test in NEW SEPARATE RUN.
4. **EACH** test should run in **max of 1 min.**

tst_placement.c (tpp): tests page faults on stack + page placement

- `FOS> run tpp 20`

tst_invalid_access.c (tia): tests handling illegal memory access (request to access page that's not exist in page file and not belong to the stack)

- `FOS> run tia 15`

tst_page_replacement_alloc.c (tpr1): tests allocation in memory and page file after page replacement

- `FOS> run tpr1 11`

tst_page_replacement_stack.c (tpr2): tests page replacement of stack (creating, modifying and reading them)

- `FOS> run tpr2 6`

tst_page_replacement_mod_clock.c (tmodclk): tests page replacement by MODIFIED CLOCK algorithm

- `FOS> run tmodclk 11`

tst_buffer_1.c (tpb1): tests page buffering and un-buffering during replacement

- `FOS> run tpb1 11`

tst_buffer_2.c (tpb2): tests freeing the modified frame list when it reaches max size

- `FOS> modbufflength 10`
- `FOS> run tpb2 11`

quicksort_heap.c (tqs): tests **running quick sort that cause memory leakage** to check the modified clock **WITH** and **WITHOUT** buffering using different strategy and different working set sizes [**EACH IN SEPARATE RUN**]. **<u>For each run do all the below steps but with a different WS sizes:</u>**

- To switch between options:
  - **`FOS> buff // to enable buffering`**
  - **`FOS> nobuff // to disable buffering`**
- **`FOS> run tqs 5   // Try all the following WS sizes:`** **[5, 50, 100, 150, 200, 250]**

- Number of Elements = **200,000**

  Initialization method : **Semi random**

  Do you want to repeat (y/n) : **n**

"At each step, the program should sort the array successfully"

APPENDICES

APPENDIX I: Page File Helper Functions

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

Pages Functions

Add a new environment page to the page file

Function declaration:

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

Description:

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

Parameters:

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

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

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

Return value:

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

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

Example:

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

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

if (ret == E_NO_PAGE_FILE_SPACE)

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

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

Function declaration:

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

Description:

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

Parameters:

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

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

Return value:

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

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

Example:

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

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

if (ret == E_PAGE_NOT_EXIST_IN_PF)

{      ...    }
```

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

Function declaration:

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

Description:

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

Parameters:

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

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

modified_page_frame_info: the Frame_Info* related to this page.

Return value:

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

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

Example:

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

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

Remove an existing environment page from the page file

Function declaration:

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

Description:

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

Parameters:

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

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

Example:

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

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

APPENDIX II: Working Set Structure & Helper Functions

Working Set Structure

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

It holds **THREE** important values about each page:

1. User virtual address of the page
2. Time stamp since the page is last referenced by the program (to be used in **LRU** replacement algorithm)
3. Empty flag to indicate whether the working set entry is empty or not

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

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

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

Figure 5: Definitions of the working set & its index inside `struct Env`

Working Set Functions

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

Print Working Set

Function declaration:

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

Description:

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

Parameters:

`e`: pointer to an environment

APPENDIX III: MLFQ Scheduler Data Structures and Helper Functions

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

Data Structures

1. Number of ready queues in the MLFQ
2. Array of ready queues: to be created and initialized later during the initialization of the MLFQ
3. Array of quantums in millisecond: to be created and initialized later during the initialization of the MLFQ

```
//[1] Ready queue(s) for the MLFQ or RR
struct Env_Queue env_ready_queues[5];
//[2] Quantum(s) in ms for each level of the ready queue(s)
uint8 quantums[5];
//[3] Number of ready queue(s)
uint8 num_of_ready_queues ;
```

Helper Functions

Print all scheduled processes

Function declaration:

```
void sched_print_all()
```

Description:

Print all processes exist in **NEW**, **READY** and **EXIT** queues

Set the CPU Quantum

Function declaration:

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

Description:

Set the CPU quantum by the quantum of the selected level

Parameters:

`quantum_in_ms`: quantum of the selected level in milliseconds

APPENDIX IV: Lists Helper Functions

IMPORTANT:

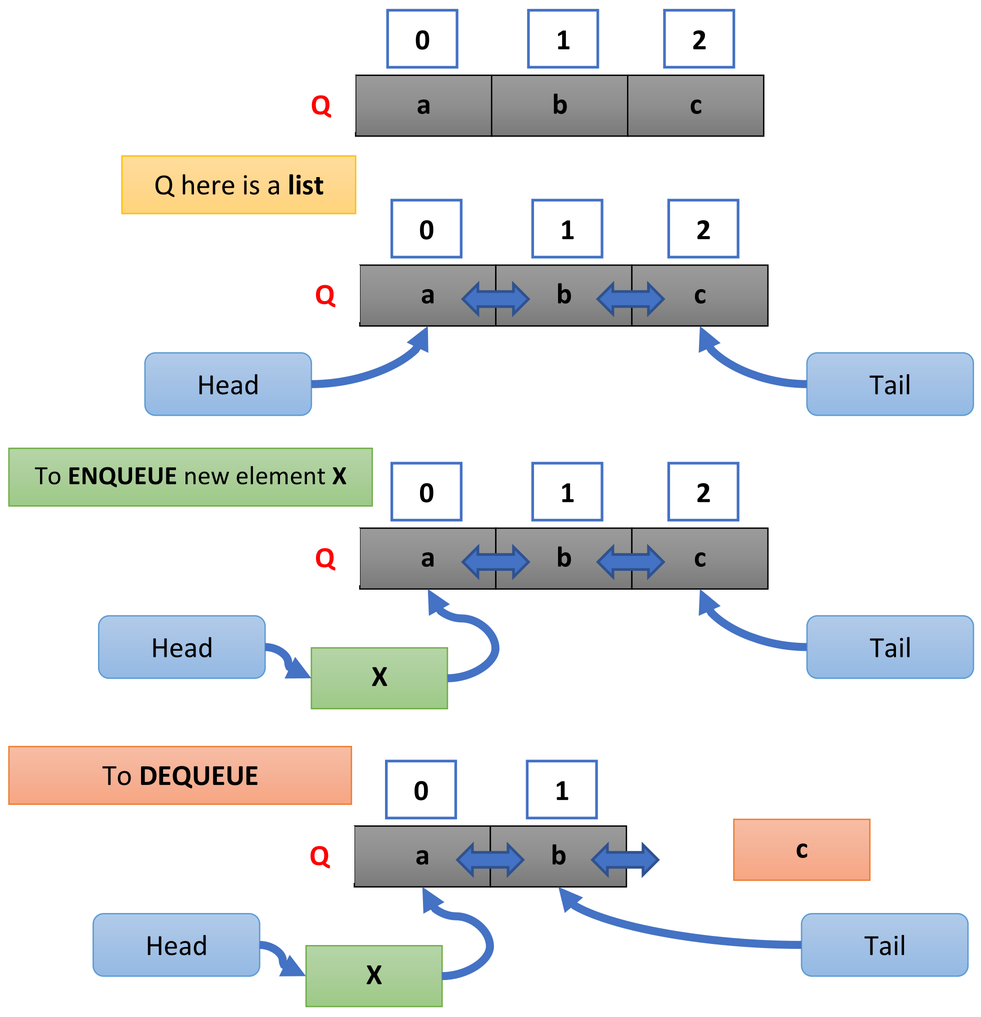
1. You should pass all the lists to the functions by reference Put **&** before the name of the list
2. If you shall deal with a **QUEUE**, consider it as a **LIST**, to enqueue insert at the head of the list (queue) and to dequeue remove from the tail of the list (queue) as detailed below.

Queue is a list

Description:

If you shall deal with a QUEUE, consider it as a LIST, to enqueue insert at the head of the list (queue) and to dequeue remove from the tail of the list (queue)

Example:



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

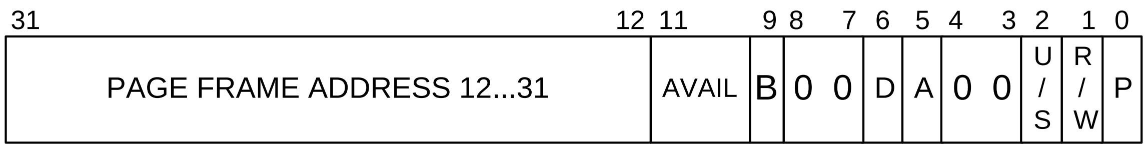
`element`: the new element to be inserted at the tail of list

APPENDIX V: Page Buffering

Structures

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

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



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

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

a- **`isBuffered`**: to indicate whether **this frame** is buffered or not

b- **`va`**: virtual address of the page that was mapped to this buffered frame

c- **`environment`**: the environment that own this virtual address

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

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

```
struct Linked_List modified_frame_list;
```

Functions

Get Maximum Length of the Modified Buffered List

Description:

Return the maximum length of the modified buffered list

Function declaration:

```
uint32 getModifiedBufferLength()
```

Parameters:

No parameters

Example:

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