ENCS3390
Operating Systems

Project
Virtual Memory Management Simulationenetics

Prepared by:

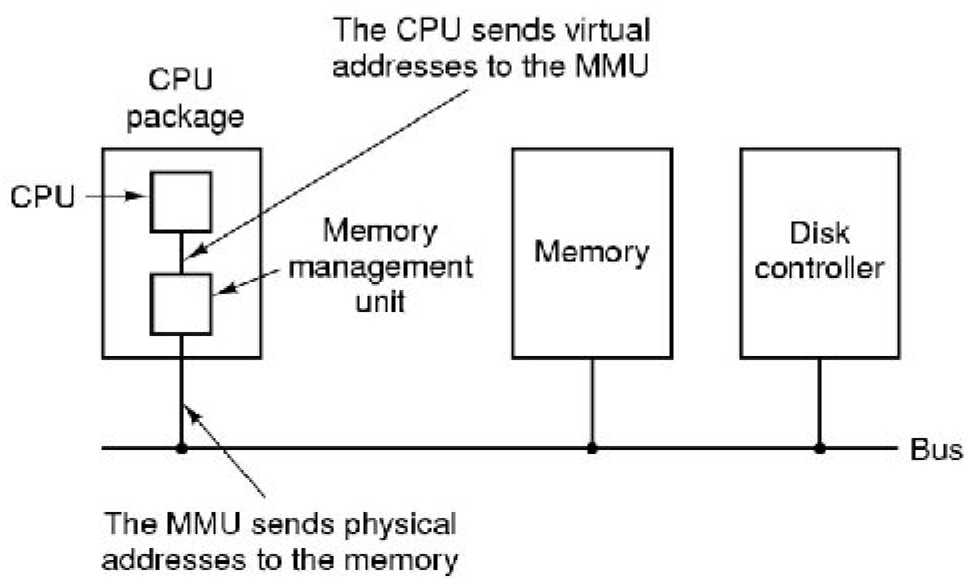
Qutaiba Olayyan	1190760
Majd Abubaha	1190069
Maysam Khatib	1190207

Instructor: Dr. Ahmad Afaneh

Section: 2, 4

Abstract

The aims of this project to implement and experiment with page replacement algorithms.



Contents

Figures

Theory

Objective:

The project's major objectives are to make a Virtual Memory Management Simulation. And to build and test a simulator for testing page replacement algorithms.
In our solution for the project, we used python language to accomplish the objectives of the project and display the results.

Introduction:

We built a paging simulator. It reads in a series of data files that indicate the page traces for particular jobs and mimics their paging needs. A random number generator is used to create the trace file, which generates random page numbers for each task. Each integer has a value that is within the known size of the program (address space).
Round Robin processes are used, page generation is constant, and the amount of memory access to each job is proportional to its length.

File Format

N // number of processes

M // size of physical memory in frames

S // minimum frames per process

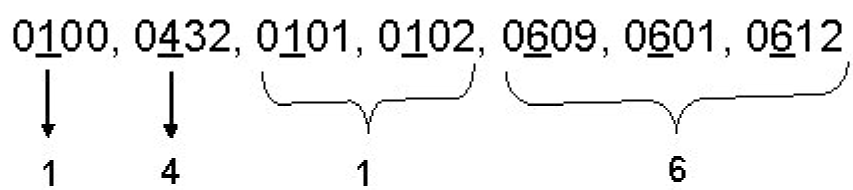
PID start Duration Size memory traces (e.g. 10 13 1A 3B ,…)

// a line for each process (N lines total)

Note: The size in the byte, and number of pages equal to size divide by 12 bit.

Memory Traces

A memory trace is a Hexadecimal list of addresses accessed by a program. By deleting the lower 12 bits, the addresses are shortened to virtual page numbers, resulting in smaller files while adhering to the process size limit.



Scheduling

The scheduling method allows programs to operate in parallel. We need to construct a round-robin amongst runnable processes with a set time Quantum for this project. The quantum is defined by the number of references observed by the simulator (a clear approximation).
Our scheduler measures the current time in cycles, with each cycle representing one memory reference in the memory trace. We assumed a cycle rate of 1000 per second.
It takes 5 cycles to switch contexts. We resume simulating memory accesses from where we left off after a context transition.
Each process' elapsed time (the period between when it started and when it finished) is reported by the scheduler. It also reports the Turnaround (TA) and Wait (W) at the end of the simulation (after all procedures have been finished).

Round Robin Example:

Process	Duration	Order	Arrival Time
P1	3	1	0
P2	4	2	0
P3	3	3	0

Suppose time quantum is 1 unit.

P1	P2	P3	P1	P2	P3	P1	P2	P3	P2

0 **10**

P1 waiting time : 4

The average waiting time(AWT) : (4+6+6)/3=5.33

P2 waiting time: 6

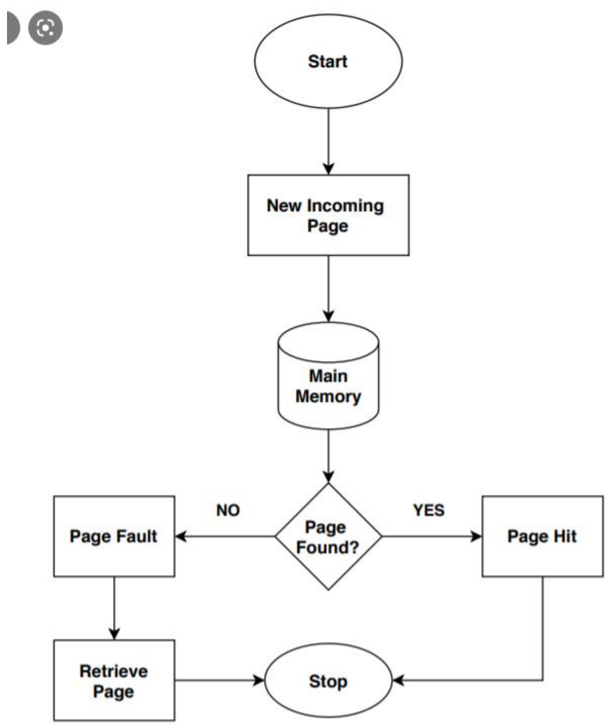
P3 waiting time: 6

Figure 1: Round Robin

Page Replacement

Our virtual machine has a certain amount of frames (physical pages) that running processes must share. We utilize disk storage when the amount of memory required exceeds the amount of physical memory accessible.
We maintained track of which pages were on disk and which were in physical memory. A page is moved from disk to memory in 300 cycles. Because this may be overlapped with processing, moving pages from memory to disk is free. We verify whether the virtual page is in physical memory for each cycle that the process runs. If that's the case, move on to the next address. If it isn't in physical memory, we mimic a page fault by context switching to a new process first, then blocking the current process, next, reading the page off disk and finally making the process ready again when the page comes back off disk.
We maintained track of which pages were in physical memory and which were on disk for each process. We requested paging because we expected that all pages begin on disk. A memory reference is not shown as a read or a write in the trace.
We provided the overall number of page faults across all processes as well as the number of page faults suffered by each process.

Page replacement policies
Based on the maximum value of the first digit of the team ID numbers, we simulated two possible page replacement policies: FIFO and LRU.
We tested both page replacement policies for each trace to see how they compared.



Round Robin Algorithm Technique

Round Robin is a CPU scheduling algorithm where each process is assigned a fixed time slot in a cyclic way.

- It is simple, easy to implement, and starvation-free as all processes get fair share of CPU.
- One of the most commonly used technique in CPU scheduling as a core.
- It is preemptive as processes are assigned CPU only for a fixed slice of time at most.
- The disadvantage of it is more overhead of context switching.

```
3
7
2
1 0 15 21000
2 2 30 6000
3 3 10 10000
```

Figure 2: Round Robin Example

Process_id	Arrival_time	Duration_time	Start_time	End_time	Turnaround_time	Waiting_time	Page_faults
1	0	15	0	18	18	8	10
2	2	30	2	5	3	1	2
3	3	10	3	14	11	5	6

Figure 3: Round Robin Example Result

FIFO Algorithm Technique

This is the simplest page replacement algorithm. In this algorithm, the operating system keeps track of all pages in the memory in a queue, the oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal.

Example: Consider these traces with 4 page frames.
Find number of page faults.

```
4C2B
3ABC
4C2B
ABFE
CBFE
BBFE
ABFE
CBFE
3ABC
9CCC
```

Figure 4: FIFO Algorithm Example

Process_id	Arrival_time	Duration_time	Start_time	End_time	Turnaround_time	Waiting_time	Page_faults
1	0	15	0	10	10	0	6

Figure 5: FIFO Algorithm Result

```
The Cycles: 1820
The Finished Time By Quantum : 10
At time 0:
The thread doesn't have any processes work on it
Ready_queue: [1]
Memory Management: ['P1_PageTable']
Memory: [0, 0, 0, 0]
At time 1:
The current process in the thread: P1
Ready_queue: []
Memory Management: ['P1_PageTable']
Memory: ['4', 0, 0, 0]
At time 2:
The current process in the thread: P1
Ready_queue: []
Memory Management: ['P1_PageTable']
Memory: ['4', '3', 0, 0]
At time 3:
The current process in the thread: P1
Ready_queue: []
Memory Management: ['P1_PageTable']
Memory: ['4', '3', 0, 0]
At time 4:
The current process in the thread: P1
Ready_queue: []
Memory Management: ['P1_PageTable']
Memory: ['4', '3', 'A', 0]
```

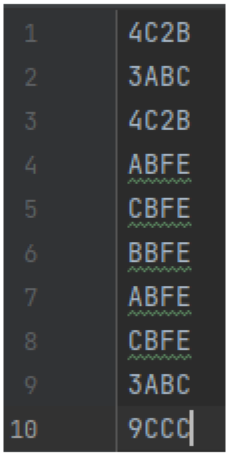
```
At time 5:
The current process in the thread: P1
Ready_queue: []
Memory Management: ['P1_PageTable']
Memory: ['4', '3', 'A', 'C']
At time 6:
The current process in the thread: P1
Ready_queue: []
Memory Management: ['P1_PageTable']
Memory: ['B', '3', 'A', 'C']
At time 7:
The current process in the thread: P1
Ready_queue: []
Memory Management: ['P1_PageTable']
Memory: ['B', '3', 'A', 'C']
At time 8:
The current process in the thread: P1
Ready_queue: []
Memory Management: ['P1_PageTable']
Memory: ['B', '3', 'A', 'C']
At time 9:
The current process in the thread: P1
Ready_queue: []
Memory Management: ['P1_PageTable']
Memory: ['B', '3', 'A', 'C']
```

```
At time 10:
The current process in the thread: P1
Ready_queue: []
Memory Management: ['P1_PageTable']
Memory: ['B', '9', 'A', 'C']
At time 11:
The thread doesn't have any processes work on it
Ready_queue: []
Memory Management: ['X']
Memory: [0, 0, 0, 0]
```

LRU Algorithm Technique

In this algorithm page will be replaced which is least recently used.

Example: Consider these traces with 4 page frames.
Find number of page faults.



```
4C2B
3ABC
4C2B
ABFE
CBFE
BBFE
ABFE
CBFE
3ABC
9CCC
```

Figure 6: LRU Algorithm Example

Process_id	Arrival_time	Duration_time	Start_time	End_time	Turnaround_time	Waiting_time	Page_faults
1	0	15	0	10	10	0	7

Figure 7: LRU Algorithm Result

```
The Cycles: 2120
The Finished Time By Quantum : 10
At time 0:
The thread doesn't have any processes work on it
Ready_queue: [1]
Memory Management: ['P1_PageTable']
Memory: [0, 0, 0, 0]
At time 1:
The current process in the thread: P1
Ready_queue: []
Memory Management: ['P1_PageTable']
Memory: ['4', 0, 0, 0]
At time 2:
The current process in the thread: P1
Ready_queue: []
Memory Management: ['P1_PageTable']
Memory: ['4', '3', 0, 0]
At time 3:
The current process in the thread: P1
Ready_queue: []
Memory Management: ['P1_PageTable']
Memory: ['4', '3', 0, 0]
At time 4:
The current process in the thread: P1
Ready_queue: []
Memory Management: ['P1_PageTable']
Memory: ['4', '3', 'A', 0]
```

```
At time 5:
The current process in the thread: P1
Ready_queue: []
Memory Management: ['P1_PageTable']
Memory: ['4', '3', 'A', 'C']
At time 6:
The current process in the thread: P1
Ready_queue: []
Memory Management: ['P1_PageTable']
Memory: ['4', 'B', 'A', 'C']
At time 7:
The current process in the thread: P1
Ready_queue: []
Memory Management: ['P1_PageTable']
Memory: ['4', 'B', 'A', 'C']
At time 8:
The current process in the thread: P1
Ready_queue: []
Memory Management: ['P1_PageTable']
Memory: ['4', 'B', 'A', 'C']
At time 9:
The current process in the thread: P1
Ready_queue: []
Memory Management: ['P1_PageTable']
Memory: ['3', 'B', 'A', 'C']
```

```
At time 10:
The current process in the thread: P1
Ready_queue: []
Memory Management: ['P1_PageTable']
Memory: ['3', '9', 'A', 'C']
At time 11:
The thread doesn't have any processes work on it
Ready_queue: []
Memory Management: ['X']
Memory: [0, 0, 0, 0]
```

Implementation

Trace Class

```python
class Trace:
    # the traces form in the files
    __ADDRESS_FORM = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 'A', 'B', 'C', 'D', 'E', 'F']

    # for save the traces of the process in this object
    # address_length = page_number
    def __init__(self, process_id, process_size, page_size_by_bits):
        self.process_id = process_id
        self.memory_traces = []
        self.address_length, self.pages_number = self.get_length_of_address_and_pages_number(process_size,
                                                                                              page_size_by_bits)
```

Figure 8: Trace Class

```python
# page_size = 12 bits, given from project
# for get the number of pages and the length of the traces
@classmethod
def get_length_of_address_and_pages_number(cls, process_size, page_size_by_bits):
    lower_digits = cls.__get_num_of_digits_by_hex_form(page_size_by_bits)
    pages_number = cls.get_pages_number(process_size, page_size_by_bits)
    number_of_pages_by_bits = cls.__get_number_of_bits_based_on_number_of_pages(pages_number)
    upper_digits = cls.__get_num_of_digits_by_hex_form(number_of_pages_by_bits)
    length_of_address = upper_digits + lower_digits
    return length_of_address, pages_number
```

Figure 9: Get Page Number from Trace Function

```python
# for create file for process traces
@classmethod
def write_memory_traces_on_file(cls, process_id, memory_trace, dictionary_path):
    file = open("{0}P{1}".format(dictionary_path, process_id), "w")
    num_of_lines = len(memory_trace)

    for i in range(num_of_lines):
        file.write(memory_trace[i])
        file.write("\n")
    file.close()

# for read the process traces file
@classmethod
def read_memory_traces_from_file(cls, process_id, dictionary_path):
    file = open("{0}P{1}".format(dictionary_path, process_id), "r")
    memory_traces = []
    for line in file.readlines():
        if line != "\n":
            memory_traces.append(line.replace("\n", ""))
    file.close()
    return memory_traces
```

Figure 10: Read and Write traces from files functions

PageTable Class

```python
class PageTable:

    def __init__(self, process_id, pages_number, process_traces):
        self.process_id = process_id  # for save this page table for which process
        self.pages_number = pages_number  # for save the number of pages of the process
        self.memory_addresses = self.get_reference_strings(process_traces)  # for save the list of the address
        self.table = HashTable(pages_number)  # for make hash table
        self.set_invalid_initially()  # for set the current pages initially invalid cuz it in the disk
```

Figure 11: Page Table Class

```python
# for change the list of traces to list of addresses
@classmethod
def get_reference_strings(cls, process_traces):
    from Trace import Trace
    from Memory import Memory
    memory_traces = []
    size = len(process_traces)
    for i in range(size):
        reference_string = Trace.get_page_number_from_trace(process_traces[i], Memory.get_page_size())
        if memory_traces.count(reference_string) == 0:
            memory_traces.append(reference_string)
    return memory_traces
```

Figure 12: convert trace to page number function

```python
# page table entry = invalid/frame_address(valid), time_when_put_in_memory, time_update_in_memory
def set_invalid_initially(self):
    for address in self.memory_addresses:
        self.table.set_val(address, [0, 0, 0])

# when send data from memory to disk
def set_invalid(self, address):
    self.table.set_val(address, [0, 0, 0])

# when get data from disk to memory
def set_valid(self, address, frame_index, time1, time2):
    self.table.set_val(address, [frame_index, time1, time2])

# update the page table
def update_page_table(self, old_address, new_address, frame_index, time1, time2):
    self.set_invalid(old_address)
    self.set_valid(new_address, frame_index, time1, time2)
```

Figure 13: set valid and invalid for pages in the pages table functions

Process Class

```python
class Process:

    __Processes_list = []  # for save the created processes

    def __init__(self, p_id, traces, size, arrival_time, duration_time):
        self.__id = p_id
        self.traces = traces  # it length equal to number of pages
        self.size = size  # in byte
        self.arrival_time = arrival_time  # the time when enter the queue
        self.duration_time = duration_time  # burst time = number of pages: each page take one unit of time
        self.state = "created"  # for save the state of the process ['ready', 'execution', 'finished']
        self.save_index = 0  # for save the last index of traces should be in the next state
        self.save_pages = []  # for save the last processing pages before enter the ready queue
        self.time = 0  # for put the time before the process enter the ready queue
        self.waiting_time = 0  # for save the times of the process in the ready queue
        self.processing_time = 0  # will increasing until reach duration_time
        self.start_time = 0  # will start when the process set in the ready queue at first time
        self.end_time = 0  # will end when the processing_time == duration_time
        self.turnaround = 0  # end_time - start_time
        self.page_faults = 0  # will increase by the page replacement algorithm
        self.exist_in_memory = False  # for check if the process has frames in the memory

        Process.insert_new_process(self)
```

Figure 14: Process Class

```python
@classmethod
def check_all_processes_finished(cls):
    done = True
    for p in cls.__Processes_list:
        if p.state != 'finished':
            done = False
            break
    return done
```

Figure 15: Check if the all processes finished function

```python
# for check if there new process came while scheduling
@classmethod
def get_new_processes(cls, arrival_time):
    arrival_processes = []
    for p in cls.__Processes_list:
        if p.arrival_time == arrival_time:
            arrival_processes.append(p)
    return arrival_processes
```

Figure 16: For insert the new arrival processes to the ready queue

```python
# when the process arrive the ready queue at first time
def enter_ready_queue_initially(self, time):
    self.start_time = time  # the first time the process inter the ready_queue
    self.enter_ready_queue(time)

def enter_ready_queue(self, time):
    self.state = 'ready'
    self.time = time  # the time when enter the ready_queue

def enter_thread(self, time):
    self.state = 'execution'
    self.waiting_time += time - self.time
    self.time = 0
```

Figure 17: Change the states of the process when enter the ready queue and leave it functions

```python
# when the process finished processing
def set_finished_if_done(self, time):
    try:
        if self.duration_time < self.processing_time:
            raise Exception
    except Exception as LD:
        print("The duration time of the P{0} finished, not enough time for all traces".format(self.__id))
        exit(-1)

    if self.processing_time == len(self.traces):
        self.state = 'finished'
        self.end_time = time
        self.set_turnaround_time()
        return True
    return False
```

Figure 18: check if the process finish and change it state function

Disk Class

```python
class Disk:
    __DISK_SIZE = 0  # TAKEN FROM USER
    __MOVE_CYCLES_FROM_DISK_TO_MEMORY = 300  # TAKEN FROM PROJECT INFO

    def __init__(self):
        self.__data_list = [0] * self.__DISK_SIZE
```

Figure 19: Disk Class

```python
def load_data_on_disk_initially(self, process_traces):
    from Trace import Trace
    from Memory import Memory
    num_of_traces = len(process_traces)
    count = 0
    for i in range(self.__DISK_SIZE):
        if count == num_of_traces:
            break
        if self.__data_list[i] == 0:
            address = Trace.get_page_number_from_trace(process_traces[count], Memory.get_page_size())
            if self.__data_list.count(address) == 0:
                self.__data_list[i] = address
            count += 1
```

Figure 20: function for load data initially in the disk

```python
# address = page number
def insert_data_by_memory_management(self, address):
    for i in range(self.__DISK_SIZE):
        if self.__data_list[i] == 0:
            self.__data_list[i] = address

# address = page number
def get_data_by_memory_management(self, address):
    for i in range(self.__DISK_SIZE):
        if self.__data_list[i] == address:
            # self.__data_list[i] = 0  # when take data from disk we didn't delete it from the disk
            return self.__data_list[i]
```

Figure 21: get and insert data in the disk functions

Memory Class

```python
class Memory:
    __MEMORY_SIZE = 0  # TAKEN FROM USER
    # the pages tables for the processes will set in the os memory size
    # each pages table for process will take just one frame
    __OPERATING_SYSTEM_SIZE = 0  # TAKEN FROM USER
    __MINIMUM_FRAMES_PER_PROCESS = 0  # TAKEN FROM USER
    __PAGE_SIZE_OF_BITS = 12  # TAKEN FROM PROJECT INFO

    def __init__(self):
        self.memory = [0] * self.__MEMORY_SIZE
```

Figure 22: Memory Class

```python
# page_table is isinstance of PageTable
def set_data_from_disk_to_memory(self, page_table, address, disk, time):
    free_frame_index = self.get_first_free_index()
    self.memory[free_frame_index] = disk.get_data_by_memory_management(address)
    page_table.set_valid(address, free_frame_index, time, time)

# page_table is isinstance of PageTable
def free_data_from_memory_to_disk(self, page_table, address):
    frame_index = page_table.table.get_val(address)[0]
    self.memory[frame_index] = 0
    page_table.set_invalid(address)

def free_frames_of_the_process(self, process):
    frames_addresses = process.save_pages
    page_table = self.get_page_table_of_process(process.get_id())
    for f in frames_addresses:
        self.free_data_from_memory_to_disk(page_table, f)
```

Figure 23: Set and free data from the memory functions

```python
# for check if we can insert new process in the memory or not
# this function used when we have many thread work as parallel
def check_minimum_frame_for_new_process(self):
    if self.memory[self.__OPERATING_SYSTEM_SIZE:].count(0) >= self.__MINIMUM_FRAMES_PER_PROCESS:
        return True
    return False
```

Figure 24: Check if there enough frames for new process function

```python
def get_frame_address_from_pages_table_by_mm(self, process_id, trace):
    from PageTable import PageTable
    from Trace import Trace
    for i in range(0, self.__OPERATING_SYSTEM_SIZE):
        if isinstance(self.memory[i], PageTable):
            if self.memory[i].process_id == process_id:
                address = Trace.get_page_number_from_trace(trace, self.__PAGE_SIZE_OF_BITS)
                return self.memory[i].table.get_val(address)[0]
    return -1  # when the address not exist at the page table
```

Figure 25: get the frame address for the process from the page table function

```python
def create_pages_table_by_mm(self, process_id, pages_number, process_traces):
    from PageTable import PageTable
    pages_table = PageTable(process_id, pages_number, process_traces)
    no_space = True

    for i in range(0, self.__OPERATING_SYSTEM_SIZE):
        if self.memory[i] == 0:
            self.memory[i] = pages_table
            no_space = False
            break

    if no_space:
        print("The memory doesn't hsa enough space for the pages table")
```

Figure 26: create pages table function

```python
# for check if the needed page for the process is in the memory or not
# if it in the memory then nothing to do
# but if it not in the memory the page replacement should work
def insert_new_page_to_process_frames_by_mm(self, process, disk, time, page_replacement):
    from PageReplacement import PageReplacement
    from Trace import Trace
    address = Trace.get_page_number_from_trace(process.traces[process.save_index], self.__PAGE_SIZE_OF_BITS)
    page_table = self.get_page_table_of_process(process.get_id())

    page_entry = page_table.table.get_val(address)
    pages_faults = PageReplacement.method(self.memory, page_table.table, address, page_entry, process.save_pages,
                                          disk, process, time, page_replacement)

    process.page_faults += pages_faults
    process.save_index += 1
    return pages_faults

# for insert the oldest pages of the process in the memory before insert the new one
def insert_old_pages_to_process_frames_by_mm(self, process, old_address, disk, time):
    page_table = self.get_page_table_of_process(process.get_id())
    frame_index = self.get_first_free_index()
    page_table.set_valid(old_address, frame_index, time, time)
    data = disk.get_data_by_memory_management(old_address)
    self.memory[frame_index] = data

    process.page_faults += 1  # get data from disk
    return 1  # return page_fault
```

Figure 27: Insert new pages in the memory by use page replacement

PageReplacment Class

```python
# page_table = hash_table
@classmethod
def method(cls, memory, page_table, address, new_page_entry, current_pages, disk, process, current_time,
           page_replacement_METHOD):
    from Memory import Memory

    if new_page_entry[0] != 0:  # the page already in the memory
        # for update the time for the page
        page_table.set_val(address, [new_page_entry[0], new_page_entry[1], current_time])

        return 0  # if the page already in the memory
    else:  # if the page in the disk not in the memory
        max_number_of_pages = Memory.get_min_frames_number()
        if len(current_pages) != max_number_of_pages:  # there is enough space for new page to put in the memory
            data_from_disk = disk.get_data_by_memory_management(address)
            frame_index = 0

            for i in range(Memory.get_sizes_info()[1], Memory.get_sizes_info()[0]):
                if memory[i] == 0:
                    frame_index = i
                    break
            memory[frame_index] = data_from_disk
            page_table.set_val(address, [frame_index, current_time, current_time])
            process.save_pages.append(address)

        else:  # when the process has max_number_of_pages
            data_from_disk = disk.get_data_by_memory_management(address)
            pages_table_entries = []
            for adds in current_pages:
                pages_table_entries.append(page_table.get_val(adds))

            victim_page_entry = 0
            if page_replacement_METHOD == PageReplacement.FIFO:
                victim_page_entry = min(pages_table_entries, key=cls.FIFO_MIN)
            elif page_replacement_METHOD == PageReplacement.LRU:
                victim_page_entry = min(pages_table_entries, key=cls.LRU_MIN)

            victim_page_address = 0
            for adds in current_pages:
                if page_table.get_val(adds) == victim_page_entry:
                    victim_page_address = adds
                    break

            frame_index = victim_page_entry[0]  # for take the frame address from old page to the new page
            memory[frame_index] = data_from_disk  # for update to new data
            page_table.set_val(victim_page_address, [0, 0, 0])
            page_table.set_val(address, [frame_index, current_time, current_time])
            process.save_pages.remove(victim_page_address)
            process.save_pages.append(address)

    return 1  # cuz we should go to disk to get the data
```

Figure 28: Page Replacement Function

Simulation Class

```python
class Simulation:
    __memory_steps = []
    __ready_queue_steps = []
    __current_processes_in_thread = []
```

Figure 29: Simulation lists

```python
@classmethod
def add_all(cls, process_id, memory, ready_queue, time):
    cls.add_process_id(process_id, time)
    cls.add_memory_step(memory, time)
    cls.add_queue_step(ready_queue, time)

@classmethod
def add_process_id(cls, process_id, time):
    cls.__current_processes_in_thread.append([time, process_id])

@classmethod
def add_memory_step(cls, memory, time):
    cls.__memory_steps.append([time, memory])

@classmethod
def add_queue_step(cls, ready_queue, time):
    processes_id = []
    for p in ready_queue:
        processes_id.append(p.get_id())
    cls.__ready_queue_steps.append([time, processes_id])
```

Figure 30: add steps in the simulation lists functions

```python
@classmethod
def simulation(cls, processing_thread):
    cycles = processing_thread.cycles
    finish_time = processing_thread.work_time
    print("The Cycles: {0}".format(cycles))
    print("The Finished Time By Quantum : {0}".format(finish_time))

    times, processes_ids, ready_queue, page_tables, frames_of_processes = cls.divide_the_data()
    for i in range(finish_time + 2):
        print("At time {0}: ".format(times[i]))
        if processes_ids[i] == "None":
            print("The thread doesn't have any processes work on it")
        else:
            print("The current process in the thread: P{0}".format(processes_ids[i]))
        print("Ready_queue: {0}".format(ready_queue[i]))
        print("Memory Management: {0}".format(page_tables[i]))
        print("Memory: {0}".format(frames_of_processes[i]))

    print("Finish The Simulation")
```

Figure 31: simulation function

CPU Class

```python
class CPU:
    _MAX_NUMBER_OF_NUMBER = 0  # READ FROM USER
    _waiting_queue = []  # WE WILL NOT USE IT CUZ NO I/O IN THIS PROJECT
    _QUANTUM = 0  # READ FROM USER
    _CONTEXT_SWITCHING_TIME_BY_CYCLE = 5  # WRITTEN IN PROJECT
```

Figure 32: CPU Parameters

```python
@classmethod
def insert_the_beginning_process_at_time_zero_to_the_ready_queue(cls, ready_queue):
    arrival_processes = Process.get_new_processes(0)
    for p in arrival_processes:
        cls.insert_process_in_ready_queue_initially(p, 0, ready_queue)

@classmethod
def insert_process_in_ready_queue_initially(cls, process, arrival_time, ready_queue):
    process.enter_ready_queue_initially(arrival_time)
    ready_queue.append(process)

@classmethod
def insert_process_in_ready_queue(cls, process, time, ready_queue):
    process.enter_ready_queue(time)
    ready_queue.append(process)
```

Figure 33: Insert process in the ready queue functions

```python
# FIFO : FIRST IN FIRST OUT : THE OLDEST PROCESS WILL BE AT INDEX[0] AND THE NEW ONE AT THE END
@classmethod
def get_process_from_ready_queue(cls, time, ready_queue):
    if len(ready_queue) == 0:
        return None
    else:
        process = ready_queue[0]
        process.enter_thread(time)
        ready_queue.remove(process)
        return process
```

Figure 34: for get the process from the ready queue to insert it in the thread

```python
@classmethod
def start_the_program(cls, file_path, dictionary_path, threads_num, quantum):
    from Memory import Memory
    from Disk import Disk
    from Thread import Thread
    # firstly for create the processes
    file = open(file_path, "r")
    all_lines = file.readlines()
    num_of_processes = int(all_lines[0])
    memory_size = int(all_lines[1])
    min_frames = int(all_lines[2])

    for i in range(3, num_of_processes + 3):
        if all_lines[i] != "\n":
            p_id, arrival_time, duration_time, p_size = list(map(int, all_lines[i].split(" ")))
            memory_traces = Trace.read_memory_traces_from_file(p_id, dictionary_path)
            Process(p_id, memory_traces, p_size, arrival_time, duration_time)
    file.close()

    # for initialize memory sizes
    Memory.set_min_framers_per_process(min_frames)
    Memory.set_memory_size(memory_size)
    Memory.set_os_size(num_of_processes)

    # for initialize disk size
    Disk.set_disk_size(10 * memory_size)

    # for initialize number of threads
    cls.set_number_of_threads(threads_num)

    # for initialize the quantum
    cls.set_quantum(quantum)

    # for create memory and disk
    memory = Memory()
    disk = Disk()

    # for crete threads for mm and disk
    mm_thread = Thread.create_thread()
    mm_thread.set_mm_in_thread(memory)
    disk_thread = Thread.create_thread()
    disk_thread.set_disk_in_thread(disk)

    # for insert all data on the disk initially
    disk_thread.thread_of_disk_insert_data_initially_in_disk()

    # for create pages table for each process
    mm_thread.thread_of_mm_create_pages_tables_initially()

    return mm_thread, disk_thread
```

Figure 35: Start Program Function

Thread Class

```python
# one thread for memory management for updating the pages tables
# and one thread for disk access
# other threads for processes


class Thread(CPU):
    __NUMBER_OF_CREATED_THREADS = 0

    def __init__(self):
        self.processing_thread = False  # if this thread used for processes
        self.process = None  # if this thread for process then self.process should equal to the process reference
        self.work_time = 0  # for save the processing time
        self.cycles = 0
        self.ready_queue = []
        self.mm_exist = False  # if this thread for memory management then mm_exist should equal True
        self.mm = None
        self.disk_exist = False  # if this thread for disk then disk_exist should equal True
        self.disk = None
```

Figure 36: Thread Class

```python
def thread_of_disk_insert_data_initially_in_disk(self):
    if self.disk_exist:
        traces_of_processes = Process.get_traces_of_processes(self)
        for traces in traces_of_processes:
            self.disk.load_data_on_disk_initially(traces)

def thread_of_mm_clear_frames_of_the_process(self, process):
    if self.mm_exist:
        self.mm.free_frames_of_the_process(process)

def thread_of_mm_clear_the_process_from_memory(self, process):
    if self.mm_exist:
        self.mm.free_frames_of_the_process(process)
        self.mm.delete_pages_table_by_mm(process.get_id())

def thread_of_mm_create_pages_tables_initially(self):
    if self.mm_exist:
        all_processes = Process.get_all_processes(self)
        for process in all_processes:
            self.mm.create_pages_table_by_mm(process.get_id(),
                                             Trace.get_pages_number(process.size, self.mm.get_page_size()),
                                             process.traces)
```

Figure 37: memory management and disk function by threads

```python
# each page reference take one quantum
# if quantum = 3, then there is 3 page references
def processing(self, mm_thread, disk_thread, page_replacement):
    if self.processing_thread:
        self.insert_the_beginning_process_at_time_zero_to_the_ready_queue(self.ready_queue)
        # insert the data for simulation
        Simulation.add_all("None", mm_thread.mm.memory.copy(), self.ready_queue, self.work_time)
        while True:

            current_process = self.get_process_from_ready_queue(self.work_time, self.ready_queue)
            if current_process is None:  # when no any process in the ready queue
                check_done = Process.check_all_processes_finished()  # check if the all processes finished
                if check_done:
                    # insert the data for simulation
                    # when clear the memory then we should put the time plus one
                    Simulation.add_all("None", mm_thread.mm.memory.copy(), self.ready_queue, self.work_time + 1)
                    break
                else:
                    self.work_time += 1  # if the processes not finished yet and not enter the ready queue
                    # for get the arrival processes and put them in the ready queue
                    arrival_processes = Process.get_new_processes(self.work_time)
                    for p in arrival_processes:
                        self.insert_process_in_ready_queue_initially(p, self.work_time, self.ready_queue)
                    # insert the data for simulation
                    Simulation.add_all("None", mm_thread.mm.memory.copy(), self.ready_queue, self.work_time)

            else:
                # put the process in the thread
                self.process = current_process
```
Figure 38: Processing Function of thread part 1

```python
else:
    # put the process in the thread
    self.process = current_process
    # Context switching should take 5 cycles
    self.cycles += self._CONTEXT_SWITCHING_TIME_BY_CYCLE
    memory_accesses = 0
    check_current_process = False
    # for insert the oldest pages in the memory before the new one
    old_pages = self.process.save_pages
    number_of_old_pages = len(old_pages)
    old_pages_counter = 0

    while memory_accesses != self._QUANTUM:

        # for put the oldest pages in the memory after make page replacement for the new page
        if not self.process.exist_in_memory and old_pages_counter != number_of_old_pages:
            page_faults = mm_thread.mm.insert_old_pages_to_process_frames_by_mm(self.process,
                                                                                 old_pages[
                                                                                     old_pages_counter],
                                                                                 disk_thread.disk,
                                                                                 self.work_time)
            old_pages_counter += 1

        else:
            page_faults = mm_thread.mm.insert_new_page_to_process_frames_by_mm(self.process,
                                                                                disk_thread.disk,
                                                                                self.work_time,
                                                                                page_replacement)
            # in the else scope
            ###
            self.process.processing_time += 1  # when insert new page in the memory
            self.process.exist_in_memory = True
            ###

        self.work_time += 1
        # memory references should take 1 cycle
        self.cycles += 1
        # disk reference should take 300 cycle
        self.cycles += page_faults * disk_thread.disk.get_search_cycles()
        memory_accesses += 1
        check_process_done = self.process.set_finished_if_done(self.work_time)
```

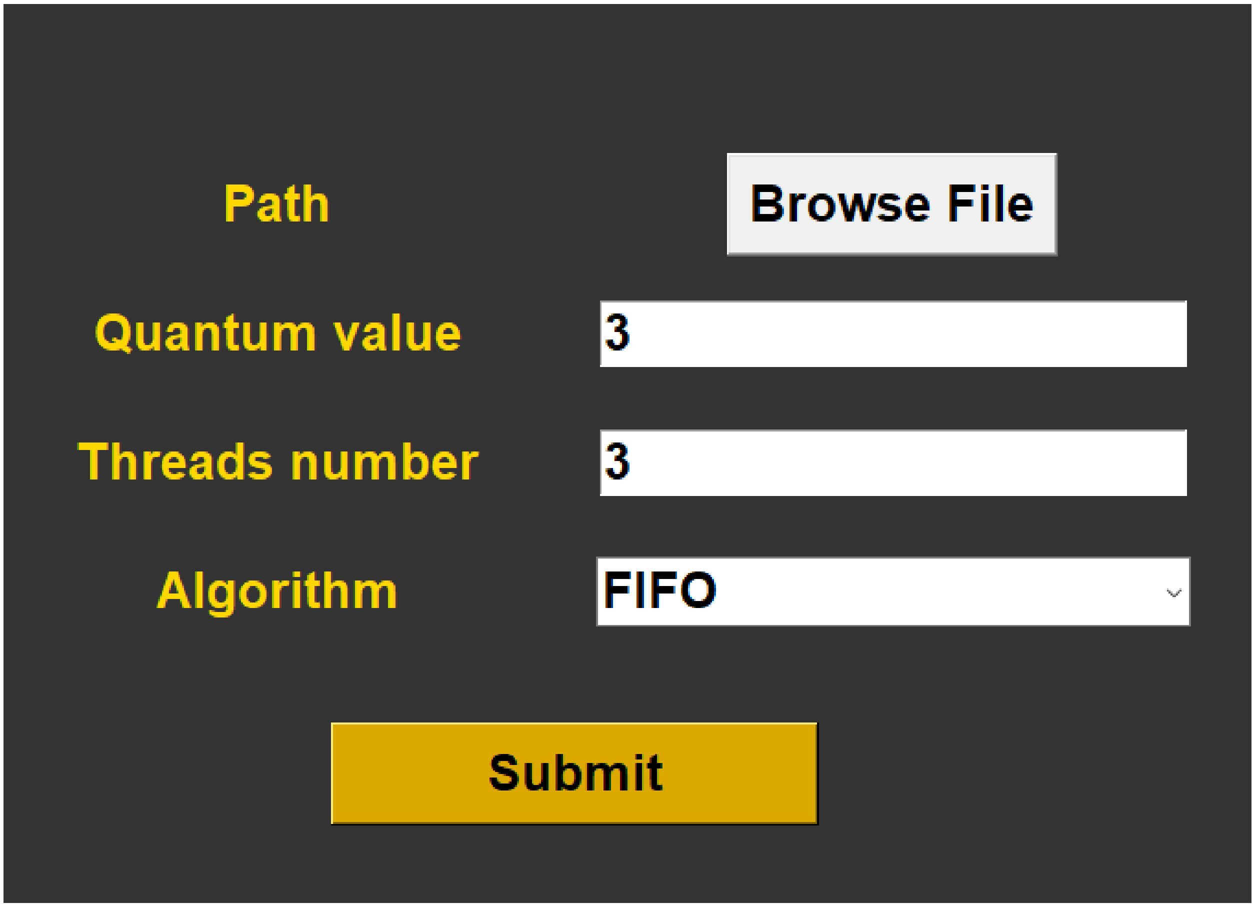
Figure 39: Processing Function of thread part 2

```python
    self.work_time += 1
    # memory references should take 1 cycle
    self.cycles += 1
    # disk reference should take 300 cycle
    self.cycles += page_faults * disk_thread.disk.get_search_cycles()
    memory_accesses += 1
    check_process_done = self.process.set_finished_if_done(self.work_time)

    # for insert the arrival processes in the ready queue
    arrival_processes = Process.get_new_processes(self.work_time)
    for p in arrival_processes:
        self.insert_process_in_ready_queue_initially(p, self.work_time, self.ready_queue)

    # insert the data for simulation
    Simulation.add_all(self.process.get_id(), mm_thread.mm.memory.copy(), self.ready_queue,
                       self.work_time)

    if check_process_done:
        # for clear the process from memory
        mm_thread.thread_of_mm_clear_the_process_from_memory(self.process)
        self.process = None
        check_current_process = True
        break

# if the current process not finish it work yet
if not check_current_process:
    self.insert_process_in_ready_queue(self.process, self.work_time, self.ready_queue)
    # if there enough frames for new process then don't delete the frames of the current process
    if not mm_thread.mm.check_minimum_frame_for_new_process() and len(self.ready_queue) > 1:
        mm_thread.thread_of_mm_clear_frames_of_the_process(self.process)
        self.process.exist_in_memory = False
    self.process = None
```

Figure 40: Processing Function of thread part 3

Testing

Exp1: For check when there is not enough space for new process then the exists process in the memory clear its frames.




```
2
5
2
1 0 15 21000
2 2 30 6000
```

Figure 41: Exp1

Process_id	Arrival_time	Duration_time	Start_time	End_time	Turnaround_time	Waiting_time	Page_faults
1	0	15	0	14	14	2	12
2	2	30	2	5	3	1	2

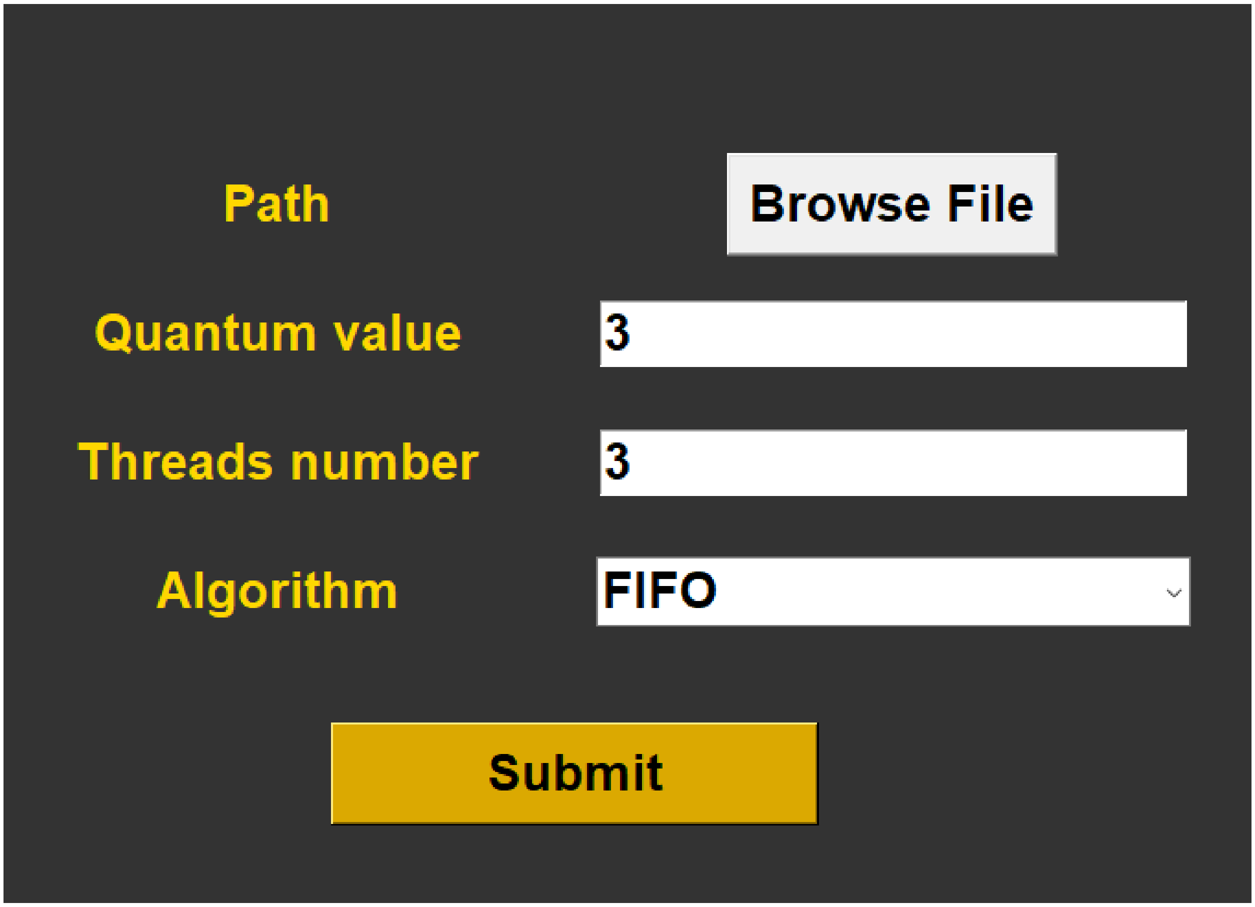
```
The Cycles: 4239
The Finished Time By Quantum : 14
At time 0:
The thread doesn't have any processes work on it
Ready_queue: [1]
Memory Management: ['P1_PageTable', 'P2_PageTable']
Memory: [0, 0, 0]
At time 1:
The current process in the thread: P1
Ready_queue: []
Memory Management: ['P1_PageTable', 'P2_PageTable']
Memory: ['2', 0, 0]
At time 2:
The current process in the thread: P1
Ready_queue: [2]
Memory Management: ['P1_PageTable', 'P2_PageTable']
Memory: ['2', '3', 0]
At time 3:
The current process in the thread: P1
Ready_queue: [2]
Memory Management: ['P1_PageTable', 'P2_PageTable']
Memory: ['4', '3', 0]
At time 4:
The current process in the thread: P2
Ready_queue: [1]
Memory Management: ['P1_PageTable', 'P2_PageTable']
Memory: ['B', 0, 0]
```

```
At time 5:
The current process in the thread: P2
Ready_queue: [1]
Memory Management: ['P1_PageTable', 'P2_PageTable']
Memory: ['B', '5', 0]
At time 6:
The current process in the thread: P1
Ready_queue: []
Memory Management: ['P1_PageTable', 'X']
Memory: ['3', 0, 0]
At time 7:
The current process in the thread: P1
Ready_queue: []
Memory Management: ['P1_PageTable', 'X']
Memory: ['3', '4', 0]
At time 8:
The current process in the thread: P1
Ready_queue: []
Memory Management: ['P1_PageTable', 'X']
Memory: ['A', '4', 0]
At time 9:
The current process in the thread: P1
Ready_queue: []
Memory Management: ['P1_PageTable', 'X']
Memory: ['A', 'C', 0]
```

```
At time 10:
The current process in the thread: P1
Ready_queue: []
Memory Management: ['P1_PageTable', 'X']
Memory: ['B', 'C', 0]
At time 11:
The current process in the thread: P1
Ready_queue: []
Memory Management: ['P1_PageTable', 'X']
Memory: ['B', 'A', 0]
At time 12:
The current process in the thread: P1
Ready_queue: []
Memory Management: ['P1_PageTable', 'X']
Memory: ['C', 'A', 0]
At time 13:
The current process in the thread: P1
Ready_queue: []
Memory Management: ['P1_PageTable', 'X']
Memory: ['C', '3', 0]
At time 14:
The current process in the thread: P1
Ready_queue: []
Memory Management: ['P1_PageTable', 'X']
Memory: ['9', '3', 0]
```

```
At time 15:
The thread doesn't have any processes work on it
Ready_queue: []
Memory Management: ['X', 'X']
Memory: [0, 0, 0]
Finish The Simulation
```

Exp2: If there process in the memory and the new process will access by thread and want to enter the memory if there enough size then we shouldn't delete the frames of the old process.



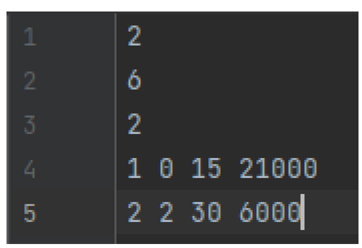


Figure 42: Exp2

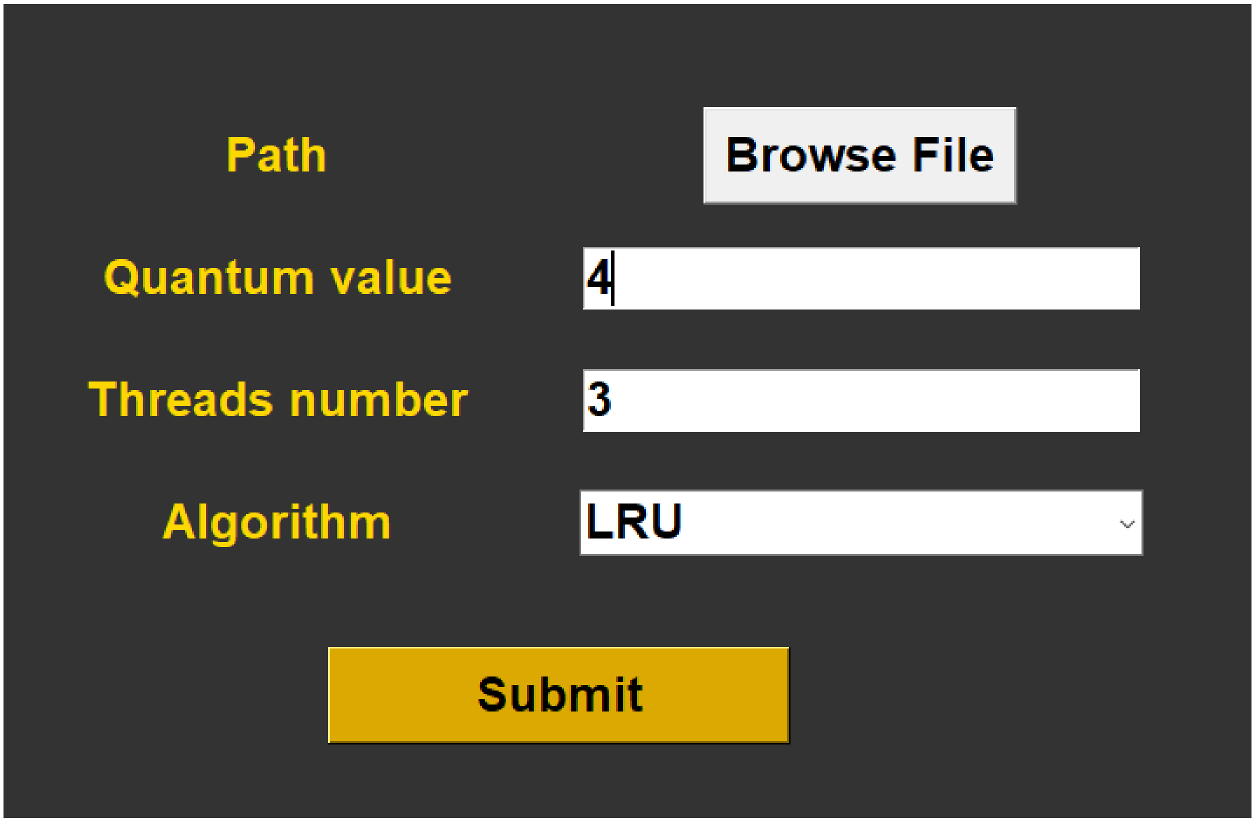
Process_id	Arrival_time	Duration_time	Start_time	End_time	Turnaround_time	Waiting_time	Page_faults
1	0	15	0	12	12	2	10
2	2	30	2	5	3	1	2

```
The Cycles: 3637
The Finished Time By Quantum : 12
At time 0:
The thread doesn't have any processes work on it
Ready_queue: [1]
Memory Management: ['P1_PageTable', 'P2_PageTable']
Memory: [0, 0, 0, 0]
At time 1:
The current process in the thread: P1
Ready_queue: []
Memory Management: ['P1_PageTable', 'P2_PageTable']
Memory: ['2', 0, 0, 0]
At time 2:
The current process in the thread: P1
Ready_queue: [2]
Memory Management: ['P1_PageTable', 'P2_PageTable']
Memory: ['2', '3', 0, 0]
At time 3:
The current process in the thread: P1
Ready_queue: [2]
Memory Management: ['P1_PageTable', 'P2_PageTable']
Memory: ['4', '3', 0, 0]
At time 4:
The current process in the thread: P2
Ready_queue: [1]
Memory Management: ['P1_PageTable', 'P2_PageTable']
Memory: ['4', '3', 'B', 0]
```

```
At time 5:
The current process in the thread: P2
Ready_queue: [1]
Memory Management: ['P1_PageTable', 'P2_PageTable']
Memory: ['4', '3', 'B', '5']
At time 6:
The current process in the thread: P1
Ready_queue: []
Memory Management: ['P1_PageTable', 'X']
Memory: ['4', 'A', 0, 0]
At time 7:
The current process in the thread: P1
Ready_queue: []
Memory Management: ['P1_PageTable', 'X']
Memory: ['C', 'A', 0, 0]
At time 8:
The current process in the thread: P1
Ready_queue: []
Memory Management: ['P1_PageTable', 'X']
Memory: ['C', 'B', 0, 0]
At time 9:
The current process in the thread: P1
Ready_queue: []
Memory Management: ['P1_PageTable', 'X']
Memory: ['A', 'B', 0, 0]
```

```
At time 10:
The current process in the thread: P1
Ready_queue: []
Memory Management: ['P1_PageTable', 'X']
Memory: ['A', 'C', 0, 0]
At time 11:
The current process in the thread: P1
Ready_queue: []
Memory Management: ['P1_PageTable', 'X']
Memory: ['3', 'C', 0, 0]
At time 12:
The current process in the thread: P1
Ready_queue: []
Memory Management: ['P1_PageTable', 'X']
Memory: ['3', '9', 0, 0]
At time 13:
The thread doesn't have any processes work on it
Ready_queue: []
Memory Management: ['X', 'X']
Memory: [0, 0, 0, 0]
Finish The Simulation
```

Exp3: If there a process not arrive yet, then the thread will wait it until arrived, then the process go through thread and go in processing.




```
3
5
2
1 0 15 21000
2 2 30 6000
3 50 10 10000
```

Figure 43: Exp3

Process_id	Arrival_time	Duration_time	Start_time	End_time	Turnaround_time	Waiting_time	Page_faults
1	0	15	0	14	14	2	12
2	2	30	2	6	4	2	2
3	50	10	50	54	4	0	4

```
The Cycles: 5443
The Finished Time By Quantum : 54
At time 0:
The thread doesn't have any processes work on it
Ready_queue: [1]
Memory Management: ['P1_PageTable', 'P2_PageTable', 'P3_PageTable']
Memory: [0, 0]
At time 1:
The current process in the thread: P1
Ready_queue: []
Memory Management: ['P1_PageTable', 'P2_PageTable', 'P3_PageTable']
Memory: ['2', 0]
At time 2:
The current process in the thread: P1
Ready_queue: [2]
Memory Management: ['P1_PageTable', 'P2_PageTable', 'P3_PageTable']
Memory: ['2', '3']
At time 3:
The current process in the thread: P1
Ready_queue: [2]
Memory Management: ['P1_PageTable', 'P2_PageTable', 'P3_PageTable']
Memory: ['4', '3']
At time 4:
The current process in the thread: P1
Ready_queue: [2]
Memory Management: ['P1_PageTable', 'P2_PageTable', 'P3_PageTable']
Memory: ['4', 'A']
```

```
At time 5:
The current process in the thread: P2
Ready_queue: [1]
Memory Management: ['P1_PageTable', 'P2_PageTable', 'P3_PageTable']
Memory: ['B', 0]
At time 6:
The current process in the thread: P2
Ready_queue: [1]
Memory Management: ['P1_PageTable', 'P2_PageTable', 'P3_PageTable']
Memory: ['B', '5']
At time 7:
The current process in the thread: P1
Ready_queue: []
Memory Management: ['P1_PageTable', 'X', 'P3_PageTable']
Memory: ['4', 0]
At time 8:
The current process in the thread: P1
Ready_queue: []
Memory Management: ['P1_PageTable', 'X', 'P3_PageTable']
Memory: ['4', 'A']
At time 9:
The current process in the thread: P1
Ready_queue: []
Memory Management: ['P1_PageTable', 'X', 'P3_PageTable']
Memory: ['C', 'A']
```

```
At time 10:
The current process in the thread: P1
Ready_queue: []
Memory Management: ['P1_PageTable', 'X', 'P3_PageTable']
Memory: ['C', 'B']
At time 11:
The current process in the thread: P1
Ready_queue: []
Memory Management: ['P1_PageTable', 'X', 'P3_PageTable']
Memory: ['A', 'B']
At time 12:
The current process in the thread: P1
Ready_queue: []
Memory Management: ['P1_PageTable', 'X', 'P3_PageTable']
Memory: ['A', 'C']
At time 13:
The current process in the thread: P1
Ready_queue: []
Memory Management: ['P1_PageTable', 'X', 'P3_PageTable']
Memory: ['3', 'C']
At time 14:
The current process in the thread: P1
Ready_queue: []
Memory Management: ['P1_PageTable', 'X', 'P3_PageTable']
Memory: ['3', '9']
```

```
At time 15:
The thread doesn't have any processes work on it
Ready_queue: []
Memory Management: ['X', 'X', 'P3_PageTable']
Memory: [0, 0]
At time 16:
The thread doesn't have any processes work on it
Ready_queue: []
Memory Management: ['X', 'X', 'P3_PageTable']
Memory: [0, 0]
At time 17:
The thread doesn't have any processes work on it
Ready_queue: []
Memory Management: ['X', 'X', 'P3_PageTable']
Memory: [0, 0]
At time 18:
The thread doesn't have any processes work on it
Ready_queue: []
Memory Management: ['X', 'X', 'P3_PageTable']
Memory: [0, 0]
At time 19:
The thread doesn't have any processes work on it
Ready_queue: []
Memory Management: ['X', 'X', 'P3_PageTable']
Memory: [0, 0]
```

```
At time 50:
The thread doesn't have any processes work on it
Ready_queue: [3]
Memory Management: ['X', 'X', 'P3_PageTable']
Memory: [0, 0]
At time 51:
The current process in the thread: P3
Ready_queue: []
Memory Management: ['X', 'X', 'P3_PageTable']
Memory: ['4', 0]
At time 52:
The current process in the thread: P3
Ready_queue: []
Memory Management: ['X', 'X', 'P3_PageTable']
Memory: ['4', 'A']
At time 53:
The current process in the thread: P3
Ready_queue: []
Memory Management: ['X', 'X', 'P3_PageTable']
Memory: ['C', 'A']
At time 54:
The current process in the thread: P3
Ready_queue: []
Memory Management: ['X', 'X', 'P3_PageTable']
Memory: ['C', 'B']
```

Exp4: For see the different between the number of page fault when the data exist in the memory and request it again with the number of page fault when the data not exist in the memory and request it from the disk.

```
2F7B
3ABC
3ABC
3ABC
3ABC
BBFE
ABFE
CBFE
3ABC
9CCC
```

Figure 44: Exp4 part 1

Process_id	Arrival_time	Duration_time	Start_time	End_time	Turnaround_time	Waiting_time	Page_faults
1	0	10	0	10	10	0	7

```
2F7B
3ABC
4C2B
ABFE
CBFE
BBFE
ABFE
CBFE
3ABC
9CCC
```

Figure 45: Exp4 part 2

Process_id	Arrival_time	Duration_time	Start_time	End_time	Turnaround_time	Waiting_time	Page_faults
1	0	10	0	10	10	0	10

Appendix

Trace Class

```
import math
import random


class Trace:
    # the traces form in the files
    __ADDRESS_FORM = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 'A', 'B', 'C', 'D', 'E',
'F']

    # for save the traces of the process in this object
    # address_length = page_number
    def __init__(self, process_id, process_size, page_size_by_bits):
        self.process_id = process_id
        self.memory_traces = []
        self.address_length, self.pages_number =
self.get_length_of_address_and_pages_number(process_size,

page_size_by_bits)

    def set_random_memory_trace(self):
        self.memory_traces =
self.create_random_memory_traces(self.address_length, self.pages_number)

    # page_size = 12 bits, given from project
    # for get the number of pages and the length of the traces
    @classmethod
    def get_length_of_address_and_pages_number(cls, process_size,
page_size_by_bits):
        lower_digits = cls.__get_num_of_digits_by_hex_form(page_size_by_bits)
        pages_number = cls.get_pages_number(process_size, page_size_by_bits)
        number_of_pages_by_bits =
cls.__get_number_of_bits_based_on_number_of_pages(pages_number)
        upper_digits =
cls.__get_num_of_digits_by_hex_form(number_of_pages_by_bits)
        length_of_address = upper_digits + lower_digits
        return length_of_address, pages_number

    # 12 bits = 3 number of hex. digits
    # each 4 bits = 1 digit of hex.
    @classmethod
    def __get_num_of_digits_by_hex_form(cls, bits):
        return math.ceil(bits / 4)

    # process size = 10000byte
    # page_size_by_bits = 12
    # page_size = 2 ^ 12 = 4096 byte
    # pages_number = ceil(10000 / 4096) = 3
```

```
    @classmethod
    def get_pages_number(cls, process_size, page_size_by_bits):
        page_size = math.pow(2, page_size_by_bits)
        pages_number = math.ceil(process_size / page_size)
        return pages_number

    # 3 pages need 2 bits
    # 5 pages need 3 bits
    # 1 or 2 pages need 1 bit
    @classmethod
    def __get_number_of_bits_based_on_number_of_pages(cls, pages_number):
        if pages_number == 1:
            return 1
        return math.ceil(math.log(pages_number) / math.log(2))

    # for create random memory traces
    @classmethod
    def create_random_memory_traces(cls, address_length, pages_number):
        memory_traces = []
        for i in range(pages_number):
            address = []
            for j in range(address_length):
                address.append(random.choice(cls.__ADDRESS_FORM))
            address = list(map(str, address))
            address = "".join(address)
            memory_traces.append(address)
        return memory_traces

    # for change the trace to address
    @classmethod
    def get_page_number_from_trace(cls, trace, page_size_by_bits):
        lower_digits = cls.__get_num_of_digits_by_hex_form(page_size_by_bits)
        upper_digits = len(trace) - lower_digits
        page_number = trace[0: upper_digits]
        return page_number

    # for create file for process traces
    @classmethod
    def write_memory_traces_on_file(cls, process_id, memory_trace,
dictionary_path):
        file = open("{0}P{1}".format(dictionary_path, process_id), "w")
        num_of_lines = len(memory_trace)

        for i in range(num_of_lines):
            file.write(memory_trace[i])
            file.write("\n")
        file.close()

    # for read the process traces file
    @classmethod
    def read_memory_traces_from_file(cls, process_id, dictionary_path):
        file = open("{0}P{1}".format(dictionary_path, process_id), "r")
        memory_traces = []
        for line in file.readlines():
            if line != "\n":
                memory_traces.append(line.replace("\n", ""))
```

```
        file.close()
        return memory_traces
```

HashTable Class

```
class HashTable:
    # Create empty bucket list of given size
    def __init__(self, size):
        self.size = size
        self.hash_table = self.create_buckets()

    def create_buckets(self):
        return [[] for _ in range(self.size)]

    # Insert values into hash map
    def set_val(self, key, val):

        # Get the index from the key
        # using hash function
        hashed_key = hash(key) % self.size

        # Get the bucket corresponding to index
        bucket = self.hash_table[hashed_key]

        found_key = False
        for index, record in enumerate(bucket):
            record_key, record_val = record

            # check if the bucket has same key as
            # the key to be inserted
            if record_key == key:
                found_key = True
                break

        # If the bucket has same key as the key to be inserted,
        # Update the key value
        # Otherwise append the new key-value pair to the bucket
        if found_key:
            bucket[index] = (key, val)
        else:
            bucket.append((key, val))

    # Return searched value with specific key
    def get_val(self, key):

        # Get the index from the key using
        # hash function
        hashed_key = hash(key) % self.size

        # Get the bucket corresponding to index
        bucket = self.hash_table[hashed_key]

        found_key = False
        for index, record in enumerate(bucket):
            record_key, record_val = record

            # check if the bucket has same key as
```

```
            # the key being searched
            if record_key == key:
                found_key = True
                break

        # If the bucket has same key as the key being searched,
        # Return the value found
        # Otherwise indicate there was no record found
        if found_key:
            return record_val
        else:
            return "No record found"

    # Remove a value with specific key
    def delete_val(self, key):

        # Get the index from the key using
        # hash function
        hashed_key = hash(key) % self.size

        # Get the bucket corresponding to index
        bucket = self.hash_table[hashed_key]

        found_key = False
        for index, record in enumerate(bucket):
            record_key, record_val = record

            # check if the bucket has same key as
            # the key to be deleted
            if record_key == key:
                found_key = True
                break
        if found_key:
            bucket.pop(index)
        return

    # To print the items of hash map
    def __str__(self):
        return "".join(str(item) for item in self.hash_table)
```

PageTable Class

```
from HashTable import HashTable


class PageTable:

    def __init__(self, process_id, pages_number, process_traces):
        self.process_id = process_id  # for save this page table for which
process
        self.pages_number = pages_number  # for save the number of pages of
the process
        self.memory_addresses = self.get_reference_strings(process_traces)  #
for save the list of the address
        self.table = HashTable(pages_number)  # for make hash table
        self.set_invalid_initially()  # for set the current pages initially
invalid cuz it in the disk

    # for change the list of traces to list of addresses
    @classmethod
    def get_reference_strings(cls, process_traces):
        from Trace import Trace
        from Memory import Memory
        memory_traces = []
        size = len(process_traces)
        for i in range(size):
            reference_string =
Trace.get_page_number_from_trace(process_traces[i], Memory.get_page_size())
            if memory_traces.count(reference_string) == 0:
                memory_traces.append(reference_string)
        return memory_traces

    # page table entry = invalid/frame_address(valid),
time_when_put_in_memory, time_update_in_memory
    def set_invalid_initially(self):
        for address in self.memory_addresses:
            self.table.set_val(address, [0, 0, 0])

    # when send data from memory to disk
    def set_invalid(self, address):
        self.table.set_val(address, [0, 0, 0])

    # when get data from disk to memory
    def set_valid(self, address, frame_index, time1, time2):
        self.table.set_val(address, [frame_index, time1, time2])

    # update the page table
    def update_page_table(self, old_address, new_address, frame_index, time1,
time2):
        self.set_invalid(old_address)
        self.set_valid(new_address, frame_index, time1, time2)
```

Process Class

```
class Process:

    __Processes_list = []  # for save the created processes

    def __init__(self, p_id, traces, size, arrival_time, duration_time):
        self.__id = p_id
        self.traces = traces  # it length equal to number of pages
        self.size = size  # in byte
        self.arrival_time = arrival_time  # the time when enter the queue
        self.duration_time = duration_time  # burst time = number of pages:
each page take one unit of time
        self.state = "created"  # for save the state of the process ['ready',
'execution', 'finished']
        self.save_index = 0  # for save the last index of traces should be in
the next state
        self.save_pages = []  # for save the last processing pages before
enter the ready queue
        self.time = 0  # for put the time before the process enter the ready
queue
        self.waiting_time = 0  # for save the times of the process in the
ready queue
        self.processing_time = 0  # will increasing until reach duration_time
        self.start_time = 0  # will start when the process set in the ready
queue at first time
        self.end_time = 0  # will end when the processing_time ==
duration_time
        self.turnaround = 0  # end_time - start_time
        self.page_faults = 0  # will increase by the page replacement
algorithm
        self.exist_in_memory = False  # for check if the process has frames
in the memory

        Process.insert_new_process(self)

    # just the mm_thread or processing_thread can access this function
    @classmethod
    def get_all_processes(cls, check_thread):
        if check_thread.mm_exist or check_thread.processing_thread:
            return cls.__Processes_list

    @classmethod
    def number_of_processes(cls):
        return len(cls.__Processes_list)

    # for insert it in the disk initially
    @classmethod
    def get_traces_of_processes(cls, disk_thread):
        if disk_thread.disk_exist:
            traces_list = []
            for p in cls.__Processes_list:
```

```
                traces_list.append(p.traces)
            return traces_list
        return None


    @classmethod
    def check_all_processes_finished(cls):
        done = True
        for p in cls.__Processes_list:
            if p.state != 'finished':
                done = False
                break
        return done

    # for check if there new process came while scheduling
    @classmethod
    def get_new_processes(cls, arrival_time):
        arrival_processes = []
        for p in cls.__Processes_list:
            if p.arrival_time == arrival_time:
                arrival_processes.append(p)
        return arrival_processes

    @classmethod
    def insert_new_process(cls, p):
        cls.__Processes_list.append(p)

    def get_id(self):
        return self.__id

    def get_current_trace(self):
        if self.save_index != len(self.traces):
            return self.traces[self.save_index]
        return -1  # when all pages done

    def set_turnaround_time(self):
        self.turnaround = self.end_time - self.start_time

    # when the process arrive the ready queue at first time
    def enter_ready_queue_initially(self, time):
        self.start_time = time  # the first time the process inter the
ready_queue
        self.enter_ready_queue(time)

    def enter_ready_queue(self, time):
        self.state = 'ready'
        self.time = time  # the time when enter the ready_queue

    def enter_thread(self, time):
        self.state = 'execution'
        self.waiting_time += time - self.time
        self.time = 0

    # when the process finished processing
    def set_finished_if_done(self, time):
        try:
```

```
            if self.duration_time < self.processing_time:
                raise Exception
        except Exception as LD:
            print("The duration time of the P{0} finished, not enough time
for all traces".format(self.__id))
            exit(-1)

        if self.processing_time == len(self.traces):
            self.state = 'finished'
            self.end_time = time
            self.set_turnaround_time()
            return True
        return False
```

Disk Class

```
class Disk:
    __DISK_SIZE = 0  # TAKEN FROM USER
    __MOVE_CYCLES_FROM_DISK_TO_MEMORY = 300  # TAKEN FROM PROJECT INFO

    def __init__(self):
        self.__data_list = [0] * self.__DISK_SIZE

    def clear_disk(self):
        for i in range(self.__DISK_SIZE):
            self.__data_list[i] = 0

    @classmethod
    def get_search_cycles(cls):
        return cls.__MOVE_CYCLES_FROM_DISK_TO_MEMORY

    @classmethod
    def set_disk_size(cls, size):
        from Memory import Memory
        if size > 0 and size >= 10 * Memory.get_sizes_info()[0]:
            cls.__DISK_SIZE = size
        else:
            cls.__DISK_SIZE = 10 * Memory.get_sizes_info()[0]
            print("At least the disk size should be 10 duplicate of the
memory size")

    @classmethod
    def get_disk_size(cls):
        return cls.__DISK_SIZE

    def load_data_on_disk_initially(self, process_traces):
        from Trace import Trace
        from Memory import Memory
        num_of_traces = len(process_traces)
        count = 0
        for i in range(self.__DISK_SIZE):
            if count == num_of_traces:
                break
            if self.__data_list[i] == 0:
                address =
Trace.get_page_number_from_trace(process_traces[count],
Memory.get_page_size())
                if self.__data_list.count(address) == 0:
                    self.__data_list[i] = address
                count += 1

    # address = page number
    def insert_data_by_memory_management(self, address):
        for i in range(self.__DISK_SIZE):
            if self.__data_list[i] == 0:
                self.__data_list[i] = address
```

```
    # address = page number
    def get_data_by_memory_management(self, address):
        for i in range(self.__DISK_SIZE):
            if self.__data_list[i] == address:
                # self.__data_list[i] = 0  # when take data from disk we
didn't delete it from the disk
                return self.__data_list[i]
```

Memory Class

```
class Memory:
    __MEMORY_SIZE = 0  # TAKEN FROM USER
    # the pages tables for the processes will set in the os memory size
    # each pages table for process will take just one frame
    __OPERATING_SYSTEM_SIZE = 0  # TAKEN FROM USER
    __MINIMUM_FRAMES_PER_PROCESS = 0  # TAKEN FROM USER
    __PAGE_SIZE_OF_BITS = 12  # TAKEN FROM PROJECT INFO

    def __init__(self):
        self.memory = [0] * self.__MEMORY_SIZE

    def get_page_table_of_process(self, process_id):
        from PageTable import PageTable
        for i in range(0, self.__OPERATING_SYSTEM_SIZE):
            if isinstance(self.memory[i], PageTable):
                if self.memory[i].process_id == process_id:
                    return self.memory[i]
        return None  # when the pages_table not exist

    def get_first_free_index(self):
        for i in range(self.__OPERATING_SYSTEM_SIZE, self.__MEMORY_SIZE):
            if self.memory[i] == 0:
                return i
        return -1  # when there no free space

    # page_table is isinstance of PageTable
    def set_data_from_disk_to_memory(self, page_table, address, disk, time):
        free_frame_index = self.get_first_free_index()
        self.memory[free_frame_index] =
disk.get_data_by_memory_management(address)
        page_table.set_valid(address, free_frame_index, time, time)

    # page_table is isinstance of PageTable
    def free_data_from_memory_to_disk(self, page_table, address):
        frame_index = page_table.table.get_val(address)[0]
        self.memory[frame_index] = 0
        page_table.set_invalid(address)

    def free_frames_of_the_process(self, process):
        frames_addresses = process.save_pages
        page_table = self.get_page_table_of_process(process.get_id())
        for f in frames_addresses:
            self.free_data_from_memory_to_disk(page_table, f)

    # for check if we can insert new process in the memory or not
    # this function used when we have many thread work as parallel
    def check_minimum_frame_for_new_process(self):
        if self.memory[self.__OPERATING_SYSTEM_SIZE:].count(0) >=
self.__MINIMUM_FRAMES_PER_PROCESS:
            return True
        return False
```

```
    def get_frame_address_from_pages_table_by_mm(self, process_id, trace):
        from PageTable import PageTable
        from Trace import Trace
        for i in range(0, self.__OPERATING_SYSTEM_SIZE):
            if isinstance(self.memory[i], PageTable):
                if self.memory[i].process_id == process_id:
                    address = Trace.get_page_number_from_trace(trace,
self.__PAGE_SIZE_OF_BITS)
                    return self.memory[i].table.get_val(address)[0]
        return -1  # when the address not exist at the page table

    def create_pages_table_by_mm(self, process_id, pages_number,
process_traces):
        from PageTable import PageTable
        pages_table = PageTable(process_id, pages_number, process_traces)
        no_space = True

        for i in range(0, self.__OPERATING_SYSTEM_SIZE):
            if self.memory[i] == 0:
                self.memory[i] = pages_table
                no_space = False
                break

        if no_space:
            print("The memory doesn't hsa enough space for the pages table")

    def delete_pages_table_by_mm(self, process_id):
        from PageTable import PageTable
        for i in range(0, self.__OPERATING_SYSTEM_SIZE):
            if isinstance(self.memory[i], PageTable):
                if self.memory[i].process_id == process_id:
                    self.memory[i] = 0

    # for check if the needed page for the process is in the memory or not
    # if it in the memory then nothing to do
    # but if it not in the memory the page replacement should work
    def insert_new_page_to_process_frames_by_mm(self, process, disk, time,
page_replacement):
        from PageReplacement import PageReplacement
        from Trace import Trace
        address =
Trace.get_page_number_from_trace(process.traces[process.save_index],
self.__PAGE_SIZE_OF_BITS)
        page_table = self.get_page_table_of_process(process.get_id())

        page_entry = page_table.table.get_val(address)
        pages_faults = PageReplacement.method(self.memory, page_table.table,
address, page_entry, process.save_pages,
                                              disk, process, time,
page_replacement)

        process.page_faults += pages_faults
        process.save_index += 1
        return pages_faults
```

```
    # for insert the oldest pages of the process in the memory before insert
the new one
    def insert_old_pages_to_process_frames_by_mm(self, process, old_address,
disk, time):
        page_table = self.get_page_table_of_process(process.get_id())
        frame_index = self.get_first_free_index()
        page_table.set_valid(old_address, frame_index, time, time)
        data = disk.get_data_by_memory_management(old_address)
        self.memory[frame_index] = data

        process.page_faults += 1  # get data from disk
        return 1  # return page_fault


    @classmethod
    def get_page_size(cls):
        return cls.__PAGE_SIZE_OF_BITS

    @classmethod
    def get_min_frames_number(cls):
        return cls.__MINIMUM_FRAMES_PER_PROCESS

    @classmethod
    def get_sizes_info(cls):
        memory_size = cls.__MEMORY_SIZE
        os_size = cls.__OPERATING_SYSTEM_SIZE
        frames_size = memory_size - os_size
        return memory_size, os_size, frames_size

    @classmethod
    def set_memory_size(cls, size):
        if size >= cls.__MINIMUM_FRAMES_PER_PROCESS +
cls.mm_size_condition():
            cls.__MEMORY_SIZE = size
        else:
            cls.__MEMORY_SIZE = cls.__MINIMUM_FRAMES_PER_PROCESS +
cls.mm_size_condition()
            print("The size of the memory should be larger than minimum
frames")

    @classmethod
    def mm_size_condition(cls):
        from Process import Process
        return Process.number_of_processes()

    @classmethod
    def set_os_size(cls, size):
        if cls.__MEMORY_SIZE > size >= cls.mm_size_condition() and
cls.__MEMORY_SIZE - size >= cls.__MINIMUM_FRAMES_PER_PROCESS:
            cls.__OPERATING_SYSTEM_SIZE = size
        else:
            cls.__OPERATING_SYSTEM_SIZE = cls.mm_size_condition()
            print("The OS SYSTEM size should be less than Memory size and
larger or equal to the number of processes")

    @classmethod
```

```python
    def set_min_framers_per_process(cls, num):
        if num >= 1:
            cls.__MINIMUM_FRAMES_PER_PROCESS = num
        else:
            cls.__MINIMUM_FRAMES_PER_PROCESS = 1
            print("The minimum frames should be larger than zero")
```

PageReplacment Class

```
class PageReplacement:
    FIFO = "FIFO"
    LRU = "LRU"

    @classmethod
    def FIFO_MIN(cls, page):
        return page[1]

    @classmethod
    def LRU_MIN(cls, page):
        return page[2]

    # page_table = hash_table
    @classmethod
    def method(cls, memory, page_table, address, new_page_entry,
current_pages, disk, process, current_time,
               page_replacement_METHOD):
        from Memory import Memory

        if new_page_entry[0] != 0:  # the page already in the memory
            # for update the time for the page
            page_table.set_val(address, [new_page_entry[0],
new_page_entry[1], current_time])

            return 0  # if the page already in the memory
        else:  # if the page in the disk not in the memory
            max_number_of_pages = Memory.get_min_frames_number()
            if len(current_pages) != max_number_of_pages:  # there is enough
space for new page to put in the memory
                data_from_disk = disk.get_data_by_memory_management(address)
                frame_index = 0

                for i in range(Memory.get_sizes_info()[1],
Memory.get_sizes_info()[0]):
                    if memory[i] == 0:
                        frame_index = i
                        break
                memory[frame_index] = data_from_disk
                page_table.set_val(address, [frame_index, current_time,
current_time])
                process.save_pages.append(address)

            else:  # when the process has max_number_of_pages
                data_from_disk = disk.get_data_by_memory_management(address)
                pages_table_entries = []
                for adds in current_pages:
                    pages_table_entries.append(page_table.get_val(adds))

                victim_page_entry = 0
                if page_replacement_METHOD == PageReplacement.FIFO:
                    victim_page_entry = min(pages_table_entries,
```

```
key=cls.FIFO_MIN)
                elif page_replacement_METHOD == PageReplacement.LRU:
                    victim_page_entry = min(pages_table_entries,
key=cls.LRU_MIN)

                victim_page_address = 0
                for adds in current_pages:
                    if page_table.get_val(adds) == victim_page_entry:
                        victim_page_address = adds
                        break

                frame_index = victim_page_entry[0]  # for take the frame
address from old page to the new page
                memory[frame_index] = data_from_disk  # for update to new
data
                page_table.set_val(victim_page_address, [0, 0, 0])
                page_table.set_val(address, [frame_index, current_time,
current_time])
                process.save_pages.remove(victim_page_address)
                process.save_pages.append(address)

        return 1  # cuz we should go to disk to get the data
```

Simulation Class

```
class Simulation:
    __memory_steps = []
    __ready_queue_steps = []
    __current_processes_in_thread = []

    @classmethod
    def add_all(cls, process_id, memory, ready_queue, time):
        cls.add_process_id(process_id, time)
        cls.add_memory_step(memory, time)
        cls.add_queue_step(ready_queue, time)

    @classmethod
    def add_process_id(cls, process_id, time):
        cls.__current_processes_in_thread.append([time, process_id])

    @classmethod
    def add_memory_step(cls, memory, time):
        cls.__memory_steps.append([time, memory])

    @classmethod
    def add_queue_step(cls, ready_queue, time):
        processes_id = []
        for p in ready_queue:
            processes_id.append(p.get_id())
        cls.__ready_queue_steps.append([time, processes_id])


    @classmethod
    def __get_list_of_ids_for_pages_tables(cls, mm_memory):
        from PageTable import PageTable
        p_ids = []
        for m in mm_memory:
            if isinstance(m, PageTable):
                p_ids.append("P{0}_PageTable".format(m.process_id))
            else:
                p_ids.append("X")  # if the frame empty
        return p_ids
    @classmethod
    def divide_the_data(cls):
        from Memory import Memory
        times = []
        page_tables = []
        frames_of_processes = []
        ready_queue = []
        processes_ids = []
        for m in cls.__memory_steps:
            times.append(m[0])

page_tables.append(cls.__get_list_of_ids_for_pages_tables(m[1][0:Memory.get_s
izes_info()[1]]))
            frames_of_processes.append(m[1][Memory.get_sizes_info()[1]:])
```

```
        for q in cls.__ready_queue_steps:
            ready_queue.append(q[1])

        for p_id in cls.__current_processes_in_thread:
            processes_ids.append(p_id[1])

        return times, processes_ids, ready_queue, page_tables,
frames_of_processes
    @classmethod
    def simulation(cls, processing_thread):
        cycles = processing_thread.cycles
        finish_time = processing_thread.work_time
        print("The Cycles: {0}".format(cycles))
        print("The Finished Time By Quantum : {0}".format(finish_time))

        times, processes_ids, ready_queue, page_tables, frames_of_processes =
cls.divide_the_data()
        for i in range(finish_time + 2):
            print("At time {0}: ".format(times[i]))
            if processes_ids[i] == "None":
                print("The thread doesn't have any processes work on it")
            else:
                print("The current process in the thread:
P{0}".format(processes_ids[i]))
            print("Ready_queue: {0}".format(ready_queue[i]))
            print("Memory Management: {0}".format(page_tables[i]))
            print("Memory: {0}".format(frames_of_processes[i]))

        print("Finish The Simulation")
```

CPU Class

```
from Process import Process
from Trace import Trace


class CPU:
    _MAX_NUMBER_OF_NUMBER = 0  # READ FROM USER
    _waiting_queue = []  # WE WILL NOT USE IT CUZ NO I/O IN THIS PROJECT
    _QUANTUM = 0  # READ FROM USER
    _CONTEXT_SWITCHING_TIME_BY_CYCLE = 5  # WRITTEN IN PROJECT

    @classmethod
    def get_max_number_of_threads(cls):
        return cls._MAX_NUMBER_OF_NUMBER

    @classmethod
    def set_number_of_threads(cls, num):
        if num > 2:
            cls._MAX_NUMBER_OF_NUMBER = num
        else:
            cls._MAX_NUMBER_OF_NUMBER = 3
            print("At least 3 threads one for memory management and disk and
the other processes")

    @classmethod
    def set_quantum(cls, q):
        from Memory import Memory
        # min_frames for put the oldest frames in the memory
        # then make replacement for the new pages
        if q >= Memory.get_min_frames_number() + 1:
            cls._QUANTUM = q
        else:
            cls._QUANTUM = Memory.get_min_frames_number() + 1
            print("the quantum should be larger than
{0}".format(Memory.get_min_frames_number()))

    @classmethod
    def insert_the_beginning_process_at_time_zero_to_the_ready_queue(cls,
ready_queue):
        arrival_processes = Process.get_new_processes(0)
        for p in arrival_processes:
            cls.insert_process_in_ready_queue_initially(p, 0, ready_queue)

    @classmethod
    def insert_process_in_ready_queue_initially(cls, process, arrival_time,
ready_queue):
        process.enter_ready_queue_initially(arrival_time)
        ready_queue.append(process)

    @classmethod
    def insert_process_in_ready_queue(cls, process, time, ready_queue):
        process.enter_ready_queue(time)
        ready_queue.append(process)
```

```
    # this function used when there a priority
    @classmethod
    def get_process_from_ready_queue_to_thread(cls, process_id, ready_queue):
        for p in ready_queue:
            if p.get_id() == process_id:
                process = p
                ready_queue.remove(p)
                process.enter_thread()
                return process

        print("The process {0} not in the ready queue".format(process_id))
        return None

    # FIFO : FIRST IN FIRST OUT : THE OLDEST PROCESS WILL BE AT INDEX[0] AND
THE NEW ONE AT THE END
    @classmethod
    def get_process_from_ready_queue(cls, time, ready_queue):
        if len(ready_queue) == 0:
            return None
        else:
            process = ready_queue[0]
            process.enter_thread(time)
            ready_queue.remove(process)
            return process

    @classmethod
    def start_the_program(cls, file_path, dictionary_path, threads_num,
quantum):
        from Memory import Memory
        from Disk import Disk
        from Thread import Thread
        # firstly for create the processes
        file = open(file_path, "r")
        all_lines = file.readlines()
        num_of_processes = int(all_lines[0])
        memory_size = int(all_lines[1])
        min_frames = int(all_lines[2])

        for i in range(3, num_of_processes + 3):
            if all_lines[i] != "\n":
                p_id, arrival_time, duration_time, p_size = list(map(int,
all_lines[i].split(" ")))
                memory_traces = Trace.read_memory_traces_from_file(p_id,
dictionary_path)
                Process(p_id, memory_traces, p_size, arrival_time,
duration_time)
        file.close()

        # for initialize memory sizes
        Memory.set_min_framers_per_process(min_frames)
        Memory.set_memory_size(memory_size)
        Memory.set_os_size(num_of_processes)

        # for initialize disk size
        Disk.set_disk_size(10 * memory_size)
```

```python
    # for initialize number of threads
    cls.set_number_of_threads(threads_num)

    # for initialize the quantum
    cls.set_quantum(quantum)

    # for create memory and disk
    memory = Memory()
    disk = Disk()

    # for crete threads for mm and disk
    mm_thread = Thread.create_thread()
    mm_thread.set_mm_in_thread(memory)
    disk_thread = Thread.create_thread()
    disk_thread.set_disk_in_thread(disk)

    # for insert all data on the disk initially
    disk_thread.thread_of_disk_insert_data_initially_in_disk()

    # for create pages table for each process
    mm_thread.thread_of_mm_create_pages_tables_initially()

    return mm_thread, disk_thread
```

Thread Class

```
from CPU import CPU
from Process import Process
from Trace import Trace
from Simulation import Simulation


# one thread for memory management for updating the pages tables
# and one thread for disk access
# other threads for processes


class Thread(CPU):
    __NUMBER_OF_CREATED_THREADS = 0

    def __init__(self):
        self.processing_thread = False  # if this thread used for processes
        self.process = None  # if this thread for process then self.process
should equal to the process reference
        self.work_time = 0  # for save the processing time
        self.cycles = 0
        self.ready_queue = []
        self.mm_exist = False  # if this thread for memory management then
mm_exist should equal True
        self.mm = None
        self.disk_exist = False  # if this thread for disk then disk_exist
should equal True
        self.disk = None

    # for not allowed to create more than THREADS_NUMBER
    @classmethod
    def create_thread(cls):
        if cls._MAX_NUMBER_OF_NUMBER <= cls.__NUMBER_OF_CREATED_THREADS:
            print("You can't create more than {0}
threads".format(cls._MAX_NUMBER_OF_NUMBER))
            return None
        cls.__NUMBER_OF_CREATED_THREADS += 1
        return cls()

    # each page reference take one quantum
    # if quantum = 3, then there is 3 page references
    def processing(self, mm_thread, disk_thread, page_replacement):
        if self.processing_thread:

self.insert_the_beginning_process_at_time_zero_to_the_ready_queue(self.ready_
queue)
            # insert the data for simulation
            Simulation.add_all("None", mm_thread.mm.memory.copy(),
self.ready_queue, self.work_time)
            while True:

                current_process =
self.get_process_from_ready_queue(self.work_time, self.ready_queue)
```

```
                if current_process is None:  # when no any process in the 
ready queue
                    check_done = Process.check_all_processes_finished()  # 
check if the all processes finished
                    if check_done:
                        # insert the data for simulation
                        # when clear the memory then we should put the time 
plus one
                        Simulation.add_all("None", 
mm_thread.mm.memory.copy(), self.ready_queue, self.work_time + 1)
                        break
                    else:
                        self.work_time += 1  # if the processes not finished 
yet and not enter the ready queue
                        # for get the arrival processes and put them in the 
ready queue
                        arrival_processes = 
Process.get_new_processes(self.work_time)
                        for p in arrival_processes:
                            self.insert_process_in_ready_queue_initially(p, 
self.work_time, self.ready_queue)
                        # insert the data for simulation
                        Simulation.add_all("None", 
mm_thread.mm.memory.copy(), self.ready_queue, self.work_time)

                else:
                    # put the process in the thread
                    self.process = current_process
                    # Context switching should take 5 cycles
                    self.cycles += self._CONTEXT_SWITCHING_TIME_BY_CYCLE
                    memory_accesses = 0
                    check_current_process = False
                    # for insert the oldest pages in the memory before the 
new one
                    old_pages = self.process.save_pages
                    number_of_old_pages = len(old_pages)
                    old_pages_counter = 0

                    while memory_accesses != self._QUANTUM:

                        # for put the oldest pages in the memory after make 
page replacement for the new page
                        if not self.process.exist_in_memory and 
old_pages_counter != number_of_old_pages:
                            page_faults = 
mm_thread.mm.insert_old_pages_to_process_frames_by_mm(self.process,

old_pages[

old_pages_counter],

disk_thread.disk,

self.work_time)
                            old_pages_counter += 1
```

```
                        else:
                            page_faults =
mm_thread.mm.insert_new_page_to_process_frames_by_mm(self.process,

disk_thread.disk,

self.work_time,

page_replacement)
                            # in the else scope
                            ###
                            self.process.processing_time += 1  # when insert
new page in the memory
                            self.process.exist_in_memory = True
                            ###

                        self.work_time += 1
                        # memory references should take 1 cycle
                        self.cycles += 1
                        # disk reference should take 300 cycle
                        self.cycles += page_faults *
disk_thread.disk.get_search_cycles()
                        memory_accesses += 1
                        check_process_done =
self.process.set_finished_if_done(self.work_time)

                        # for insert the arrival processes in the ready queue
                        arrival_processes =
Process.get_new_processes(self.work_time)
                        for p in arrival_processes:
                            self.insert_process_in_ready_queue_initially(p,
self.work_time, self.ready_queue)

                        # insert the data for simulation
                        Simulation.add_all(self.process.get_id(),
mm_thread.mm.memory.copy(), self.ready_queue,
                                           self.work_time)

                        if check_process_done:
                            # for clear the process from memory

mm_thread.thread_of_mm_clear_the_process_from_memory(self.process)
                            self.process = None
                            check_current_process = True
                            break

                    # if the current process not finish it work yet
                    if not check_current_process:
                        self.insert_process_in_ready_queue(self.process,
self.work_time, self.ready_queue)
                        # if there enough frames for new process then don't
delete the frames of the current process
                        if not
mm_thread.mm.check_minimum_frame_for_new_process() and len(self.ready_queue)
> 1:
```

```
mm_thread.thread_of_mm_clear_frames_of_the_process(self.process)
                            self.process.exist_in_memory = False
                        self.process = None

    # when create processing thread
    def set_thread_for_processes(self):
        self.processing_thread = True

    # for let the access of memory by the thread
    def set_mm_in_thread(self, memory):
        self.mm_exist = True
        self.mm = memory

    # for let the access of disk by the thread
    def set_disk_in_thread(self, disk):
        self.disk_exist = True
        self.disk = disk

    def thread_of_disk_insert_data_initially_in_disk(self):
        if self.disk_exist:
            traces_of_processes = Process.get_traces_of_processes(self)
            for traces in traces_of_processes:
                self.disk.load_data_on_disk_initially(traces)

    def thread_of_mm_clear_frames_of_the_process(self, process):
        if self.mm_exist:
            self.mm.free_frames_of_the_process(process)

    def thread_of_mm_clear_the_process_from_memory(self, process):
        if self.mm_exist:
            self.mm.free_frames_of_the_process(process)
            self.mm.delete_pages_table_by_mm(process.get_id())

    def thread_of_mm_create_pages_tables_initially(self):
        if self.mm_exist:
            all_processes = Process.get_all_processes(self)
            for process in all_processes:
                self.mm.create_pages_table_by_mm(process.get_id(),

Trace.get_pages_number(process.size, self.mm.get_page_size()),
                                                 process.traces)
```

Main File

```
from CPU import CPU
from Process import Process
from Thread import Thread
from Simulation import Simulation
from tkinter import *
from tkinter import ttk
from tkinter import filedialog
from pathlib import Path
from PageReplacement import PageReplacement

# THE TITLES FOR THE CREATED TABLE
titles = ("Process_id", "Arrival_time", "Duration_time", "Start_time",
"End_time", "Turnaround_time",
          "Waiting_time", "Page_faults")


# FOR CREATE GUI TABLE
class Table:

    # frame: THE WINDOW WHICH WE WANT TO SHOW THE TABLE ON IT
    def __init__(self, frame):
        # FOR INSERT THE TITLES AS THE FIRST ROW IN THE TABLE
        # AND MAKE 7 COLUMNS
        for i in range(len(titles)):
            x = Label(frame, text=titles[i], borderwidth=1, relief="solid",
width=20, fg='#ffffff', bg="#009879",
                      font=('tajawal', 10, 'bold'))
            x.grid(row=0, column=i)

        all_processes = Process.get_all_processes(processing_threads[0])
        table_rows = []
        for process in all_processes:
            table_rows.append(
                [process.get_id(), process.arrival_time,
process.duration_time, process.start_time, process.end_time,
                 process.turnaround,
                 process.waiting_time, process.page_faults])
        # FOR INSERT THE PROCESSES INFORMATION AND THE Scheduling RESULTS AS
ROWS IN THE TABLE
        for i in range(len(all_processes)):
            for j in range(len(titles)):
                if i % 2 == 0:
                    self.e = Label(frame, text=table_rows[i][j],
borderwidth=1, relief="solid", width=20, fg='black',
                                   bg="#f3f3f3", font=('tajawal', 10,
'bold'))
                else:
                    self.e = Label(frame, text=table_rows[i][j],
borderwidth=1, relief="solid", width=20, fg='black',
                                   bg="#dddddd", font=('tajawal', 10,
'bold'))
```

```
                self.e.grid(row=i + 1, column=j)


file_path = ""  # for save the path of the data file
dictionary_path = ""  # for save the dictionary which save all processes
traces
quantum_value = 0  # for read the quantum from user
threads_number = 0  # for read the number of threads from user
page_replacement_algorithm = ""  # for choose the page replacement algorithm
from user
correct_file = False  # if the user choose correct file


def browse_files():
    global file_path, correct_file, dictionary_path
    filename = filedialog.askopenfilename(initialdir="/",
                                          title="Select a File",
                                          filetypes=(("Text files",
                                                      "*.txt*"),
                                                     ("all files",
                                                      "*.*")))

    try:
        path = str(filename)
        my_file = Path(path)
        if not my_file.is_file():
            raise FileNotFoundError
        file_path = path
        dictionary_path = "/".join(file_path.split("/")[:-1]) + "/"
        correct_file = True

    except FileNotFoundError:
        correct_file = False
        print("The File Not Exist")


# SUBMIT PARAMETERS BUTTON FUNCTION
def save_parameters():
    global quantum_value, threads_number, page_replacement_algorithm
    end = True
    try:
        quantum = int(dynamic_quantum.get())
        if quantum <= 1:
            raise Exception
        quantum_value = quantum
    except Exception:
        end = False
        dynamic_quantum.set("At least the minimum frames + 1")
    try:
        threads = int(dynamic_threads.get())
        if threads <= 2:
            raise Exception
        threads_number = threads
    except Exception:
        end = False
        dynamic_threads.set("At least 3 threads")
    try:
```

```
        algorithm = str(dynamic_algorithm.get())
        if algorithm != PageReplacement.FIFO and algorithm !=
PageReplacement.LRU:
            raise Exception
        page_replacement_algorithm = algorithm
    except Exception:
        end = False
        dynamic_algorithm.set("Wrong")

    if end and correct_file:
        parameters_window.destroy()


# PARAMETERS WINDOW
parameters_window = Tk()
parameters_window.configure(bg='#0B2F3A')
parameters_window.minsize(300, 300)
parameters_window.title("Define Parameters for The Virtual Memory Management
Simulation")
parameters_window.resizable(width=0, height=0)

# DYNAMIC TEXT FIELD VALUES WITH DEFAULT VALUES
dynamic_quantum = IntVar(parameters_window, 3)
dynamic_threads = IntVar(parameters_window, 3)
dynamic_algorithm = StringVar(parameters_window)

# MAIN FRAME CONTAIN ALL THE LABELS AND BUTTON AND TEXT FIELDS
MainFrame = Frame(parameters_window, bd=100, width=1050, height=700,
bg="#333333")
MainFrame.grid()

path_label = Label(MainFrame, width=20, height=2, text="Path", fg="gold",
bg="#333333",
                   font=('tajawal', 20, 'bold')).grid(row=1, column=1)
button_explore = Button(MainFrame, width=10, font=('tajawal', 20, 'bold'),
                        text="Browse File", command=browse_files).grid(row=1,
column=2)

quantum_label = Label(MainFrame, width=0, height=2, text="Quantum value",
fg="gold", bg="#333333",
                      font=('tajawal', 20, 'bold')).grid(row=2, column=1)
quantum_text = Entry(MainFrame, width=21, textvariable=dynamic_quantum,
font=('tajawal', 20, 'bold')).grid(row=2,

column=2)

threads_label = Label(MainFrame, width=20, height=2, text="Threads number",
fg="gold", bg="#333333",
                      font=('tajawal', 20, 'bold')).grid(row=3, column=1)
threads_text = Entry(MainFrame, width=21, textvariable=dynamic_threads,
font=('tajawal', 20, 'bold')).grid(row=3,

column=2)

algorithm_label = Label(MainFrame, width=20, height=2, text="Algorithm",
fg="gold", bg="#333333",
```

```
                        font=('tajawal', 20, 'bold')).grid(row=4, column=1)

algorithm_box = ttk.Combobox(MainFrame, width=20, font=('tajawal', 20,
'bold'),
                             textvariable=dynamic_algorithm)

# Adding combobox drop down list
algorithm_box['values'] = ('LRU', 'FIFO')

algorithm_box.grid(row=4, column=2)

# Shows february as a default value
algorithm_box.current(1)

Label(MainFrame, height=2, bg="#333333").grid(row=5, column=1, columnspan=2)

button = Button(MainFrame, width=15, text="Submit", command=save_parameters,
fg="black", bg="#DBA901",
                font=('tajawal', 20, 'bold')).grid(row=6, column=1,
columnspan=2)

# CONTINUE THE LOOP OF PARAMETERS WINDOW UNTIL THE USER ENTER VALIDATED DATA
parameters_window.mainloop()

# for start the program
mm_thread, disk_thread = CPU.start_the_program(file_path, dictionary_path,
threads_number, quantum_value)
processing_threads = []

# CPU.get_max_number_of_threads() - 2 ; dynamic threads number
for i in range(1):
    processing_threads.append(Thread.create_thread())
    processing_threads[i].set_thread_for_processes()

for thread in processing_threads:
    thread.processing(mm_thread, disk_thread, page_replacement_algorithm)
    # for print the simulation at console
    Simulation.simulation(thread)

# for create the Scheduling Table
table_window = Tk()
table_window.title("Scheduling Results")
table_window.minsize(1315, 300)
table_window.resizable(width=0, height=0)

# Create A Main Frame
main_frame = Frame(table_window)
main_frame.pack(fill=BOTH, expand=1)

# Create A Canvas
my_canvas = Canvas(main_frame)
my_canvas.pack(side=LEFT, fill=BOTH, expand=1)

# Add A Scrollbar To The Canvas
my_scrollbar = Scrollbar(main_frame, orient=VERTICAL,
command=my_canvas.yview)
```

```
my_scrollbar.pack(side=RIGHT, fill=Y)

# Configure The Canvas
my_canvas.configure(yscrollcommand=my_scrollbar.set)
my_canvas.bind('<Configure>', lambda e:
my_canvas.configure(scrollregion=my_canvas.bbox("all")))

# Create ANOTHER Frame INSIDE the Canvas
second_frame = Frame(my_canvas)

# Add that New frame To a Window In The Canvas
my_canvas.create_window((0, 0), window=second_frame, anchor="nw")

# FOR CREATE TABLE DEPEND ON THE BEST RESULT OF THE ALGORITHMS
t = Table(second_frame)

# END THE PROGRAM WHEN CLOSE THE TABLE
table_window.mainloop()
```