

**ENCS3390**

**Operating Systems**

**Project**

**Virtual Memory Management Simulationenetics**

**Prepared by:**

Qutaiba Olayyan 1190760

Majd Abubaha 1190069

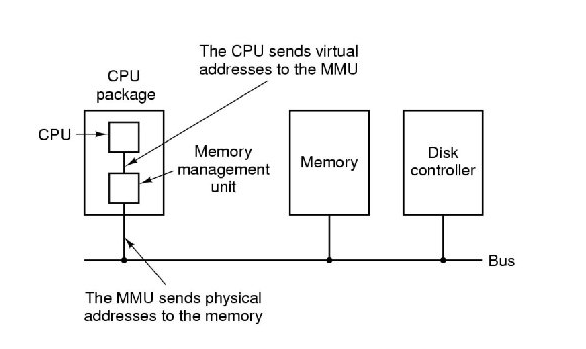
Maysam Khatib 1190207

**Instructor**: Dr. Ahmad Afaneh

**Section:** 2, 4

# Abstract

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



# Contents

[Abstract I](#_Toc92572919)

[Contents II](#_Toc92572920)

[Figures III](#_Toc92572921)

[Theory 1](#_Toc92572922)

[File Format 2](#_Toc92572923)

[Memory Traces 2](#_Toc92572924)

[Scheduling 3](#_Toc92572925)

[Page Replacement 4](#_Toc92572926)

[Round Robin Algorithm Technique 5](#_Toc92572927)

[FIFO Algorithm Technique 6](#_Toc92572928)

[LRU Algorithm Technique 9](#_Toc92572929)

[Implementation 12](#_Toc92572930)

[Trace Class 12](#_Toc92572931)

[PageTable Class 13](#_Toc92572932)

[Process Class 15](#_Toc92572933)

[Disk Class 18](#_Toc92572934)

[Memory Class 19](#_Toc92572935)

[PageReplacment Class 22](#_Toc92572936)

[Simulation Class 23](#_Toc92572937)

[CPU Class 25](#_Toc92572938)

[Thread Class 27](#_Toc92572939)

[Testing 31](#_Toc92572940)

[Exp1: For check when there is not enough space for new process then the exists process in the memory clear its frames. 31](#_Toc92572941)

[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. 35](#_Toc92572942)

[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. 39](#_Toc92572943)

[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. 45](#_Toc92572944)

[Appendix 46](#_Toc92572945)

[Trace Class 46](#_Toc92572946)

[HashTable Class 49](#_Toc92572947)

[PageTable Class 51](#_Toc92572948)

[Process Class 52](#_Toc92572949)

[Disk Class 55](#_Toc92572950)

[Memory Class 57](#_Toc92572951)

[PageReplacment Class 61](#_Toc92572952)

[Simulation Class 63](#_Toc92572953)

[CPU Class 65](#_Toc92572954)

[Thread Class 68](#_Toc92572955)

[Main File 72](#_Toc92572956)

# 

# Figures

[Figure 1: Round Robin 3](#_Toc92573002)

[Figure 2: Round Robin Example 5](#_Toc92573003)

[Figure 3: Round Robin Example Result 5](#_Toc92573004)

[Figure 4: FIFO Algorithm Example 6](#_Toc92573005)

[Figure 5: FIFO Algorithm Result 6](#_Toc92573006)

[Figure 6: LRU Algorithm Example 9](#_Toc92573007)

[Figure 7: LRU Algorithm Result 9](#_Toc92573008)

[Figure 8: Trace Class 12](#_Toc92573009)

[Figure 9: Get Page Number from Trace Function 12](#_Toc92573010)

[Figure 10: Read and Write traces from files functions 13](#_Toc92573011)

[Figure 11: Page Table Class 13](#_Toc92573012)

[Figure 12: convert trace to page number function 14](#_Toc92573013)

[Figure 13: set valid and invalid for pages in the pages table functions 14](#_Toc92573014)

[Figure 14: Process Class 15](#_Toc92573015)

[Figure 15: Check if the all processes finished function 15](#_Toc92573016)

[Figure 16: For insert the new arrival processes to the ready queue 16](#_Toc92573017)

[Figure 17: Change the states of the process when enter the ready queue and leave it functions 16](#_Toc92573018)

[Figure 18: check if the process finish and change it state fucntion 17](#_Toc92573019)

[Figure 19: Disk Class 18](#_Toc92573020)

[Figure 20: function for load data initially in the disk 18](#_Toc92573021)

[Figure 21: get and insert data in the disk functions 18](#_Toc92573022)

[Figure 22: Memory Class 19](#_Toc92573023)

[Figure 23: Set and free data from the memory functions 19](#_Toc92573024)

[Figure 24: Check if there enough frames for new process function 20](#_Toc92573025)

[Figure 25: get the frame address for the process from the page table function 20](#_Toc92573026)

[Figure 26: create pages table function 20](#_Toc92573027)

[Figure 27: Insert new pages in the memory by use page replacement 21](#_Toc92573028)

[Figure 28: Page Replacement Function 22](#_Toc92573029)

[Figure 29: Simulation lists 23](#_Toc92573030)

[Figure 30: add steps in the simulation lists functions 23](#_Toc92573031)

[Figure 31: simulation function 24](#_Toc92573032)

[Figure 32: CPU Parameters 25](#_Toc92573033)

[Figure 33: Insert process in the ready queue functions 25](#_Toc92573034)

[Figure 34: for get the process from the ready queue to insert it in the thread 25](#_Toc92573035)

[Figure 35: Start Program Function 26](#_Toc92573036)

[Figure 36: Thread Class 27](#_Toc92573037)

[Figure 37: memory management and disk function by threads 27](#_Toc92573038)

[Figure 38: Processing Function of thread part 1 28](#_Toc92573039)

[Figure 39: Processing Function of thread part 2 29](#_Toc92573040)

[Figure 40: Processing Function of thread part 3 30](#_Toc92573041)

[Figure 41: Exp1 31](#_Toc92573042)

[Figure 42: Exp2 35](#_Toc92573043)

[Figure 43: Exp3 39](#_Toc92573044)

[Figure 44: Exp4 part 1 45](#_Toc92573045)

[Figure 45: Exp4 part 2 45](#_Toc92573046)

# 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.

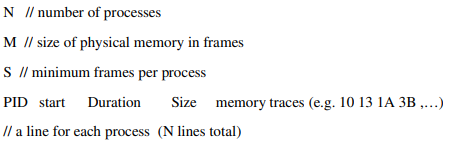
In our solution for the project, we used python language to accomplish the objectives of the project and display the results.

**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.

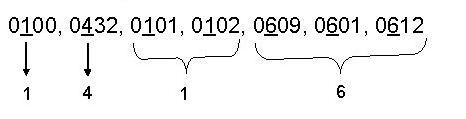
## File Format



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).

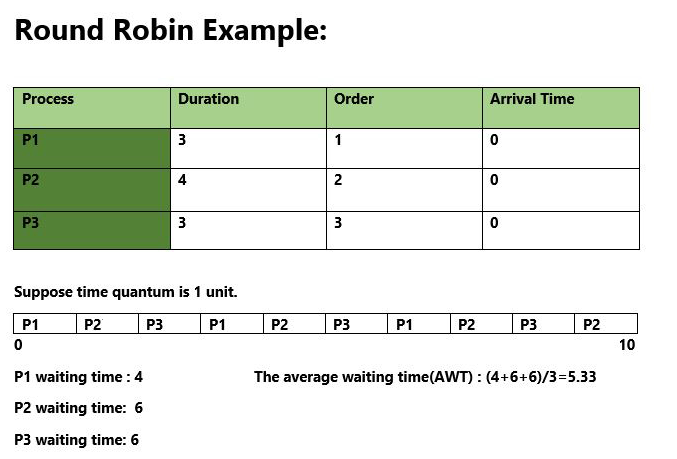


Figure 1: Round Robin

## 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.

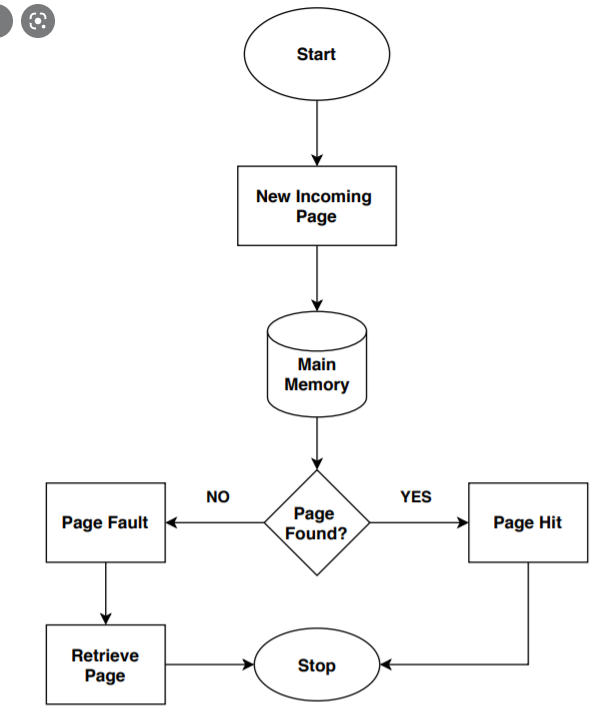
We maintained track of which pages were in physical memory and which were on disk for each process. We requested paging because we expected that all pages begin on disk. A memory reference is not shown as a read or a write in the trace.

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.

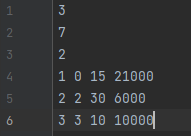


Figure 2: Round Robin Example

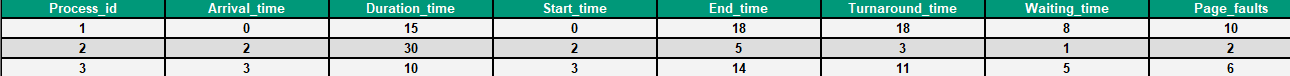


Figure 3: Round Robin Example Result

# 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.

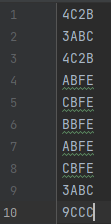
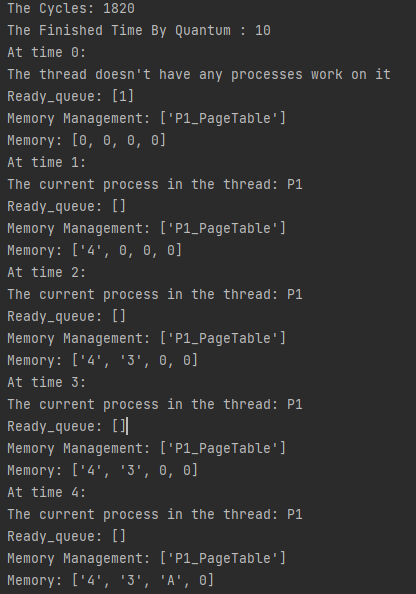
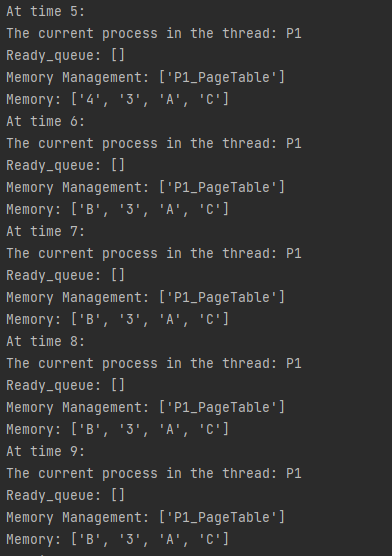


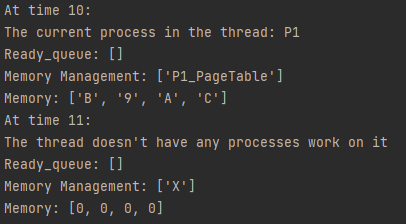
Figure 4: FIFO Algorithm Example



Figure 5: FIFO Algorithm Result







# 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.

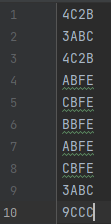
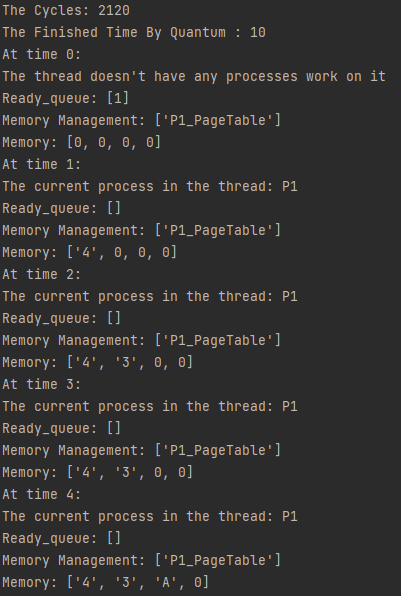
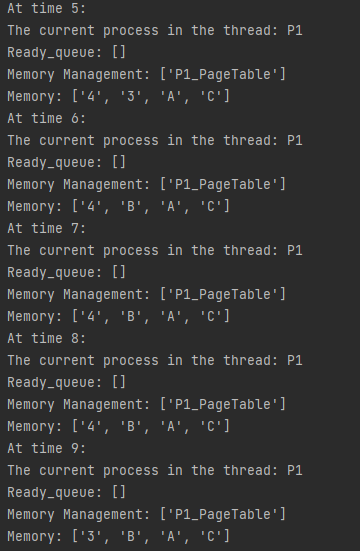


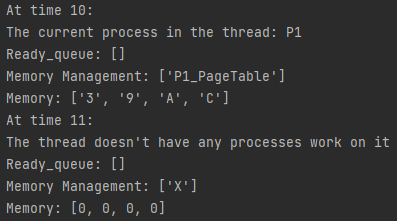
Figure 6: LRU Algorithm Example



Figure 7: LRU Algorithm Result







# Implementation

## Trace Class

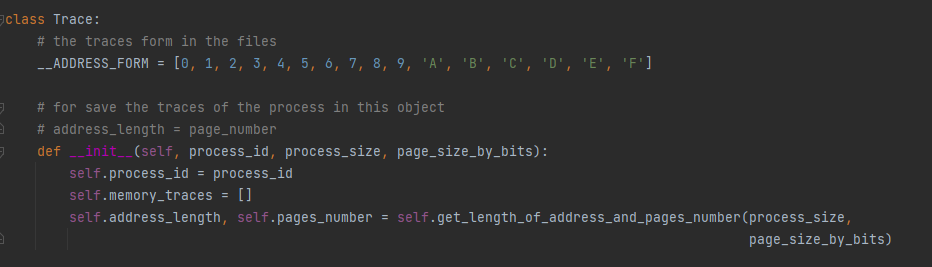


Figure 8: Trace Class

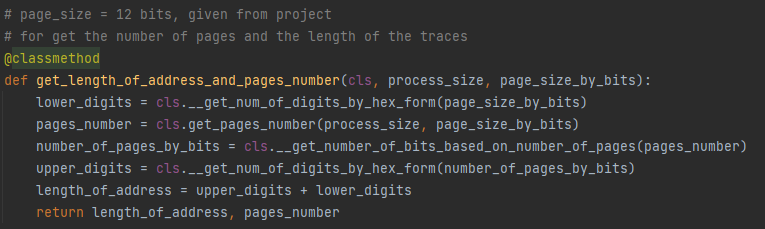


Figure 9: Get Page Number from Trace Function

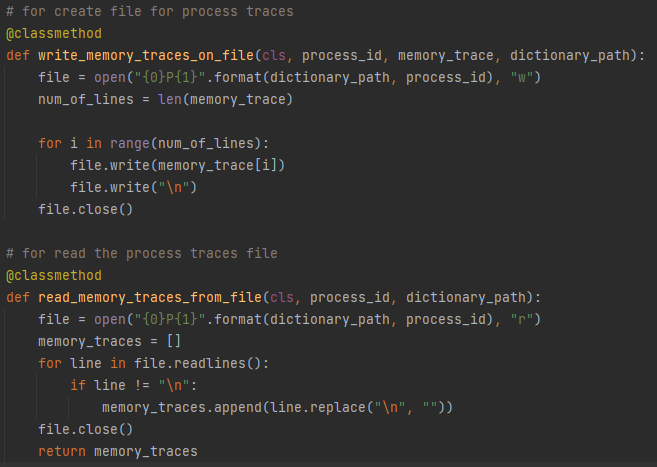


Figure 10: Read and Write traces from files functions

## PageTable Class

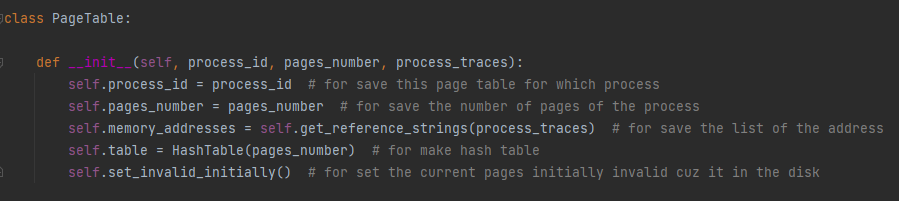


Figure 11: Page Table Class

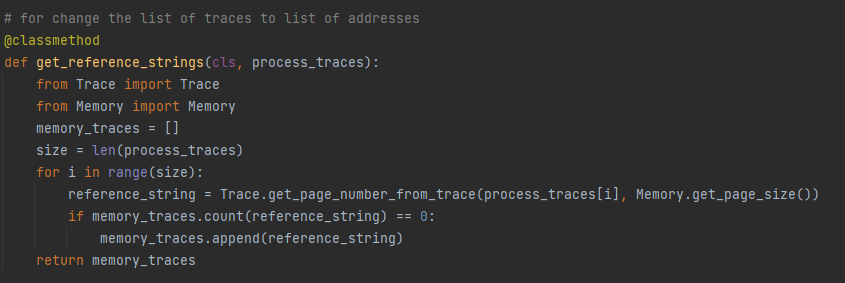


Figure 12: convert trace to page number function

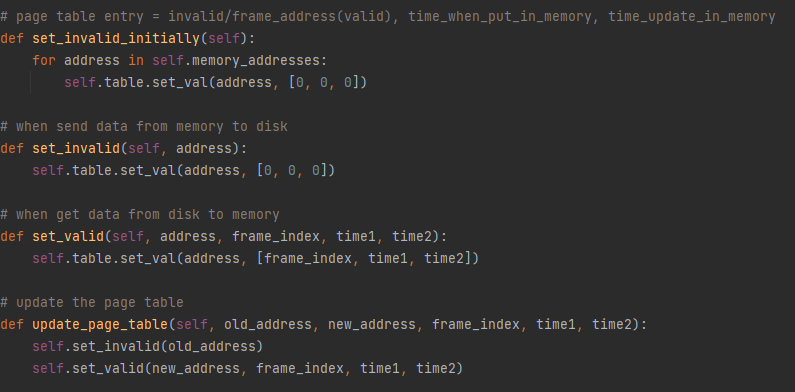


Figure 13: set valid and invalid for pages in the pages table functions

## Process Class

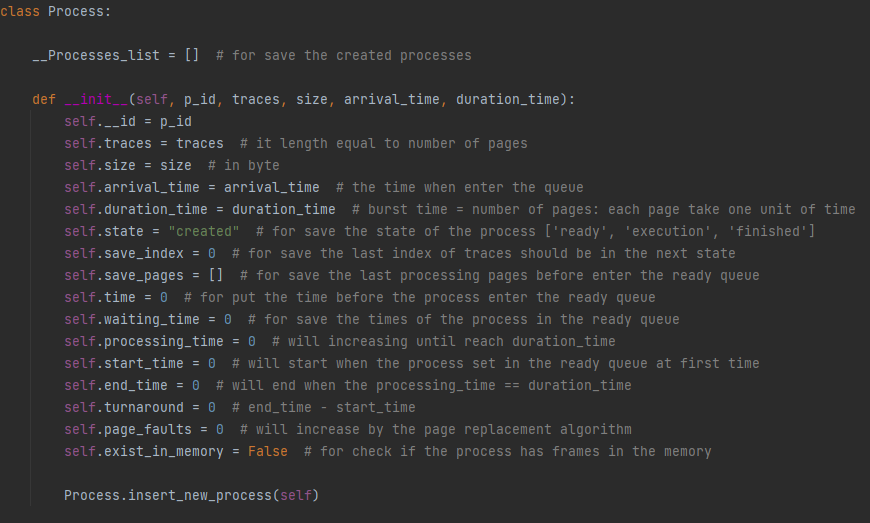


Figure 14: Process Class

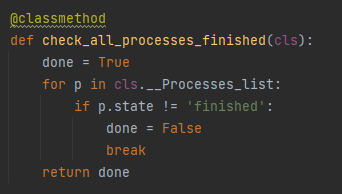


Figure 15: Check if the all processes finished function

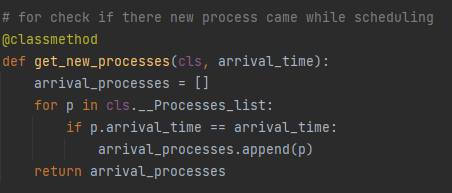


Figure 16: For insert the new arrival processes to the ready queue

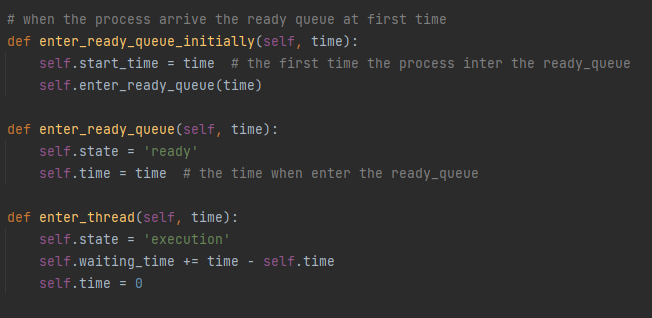


Figure 17: Change the states of the process when enter the ready queue and leave it functions

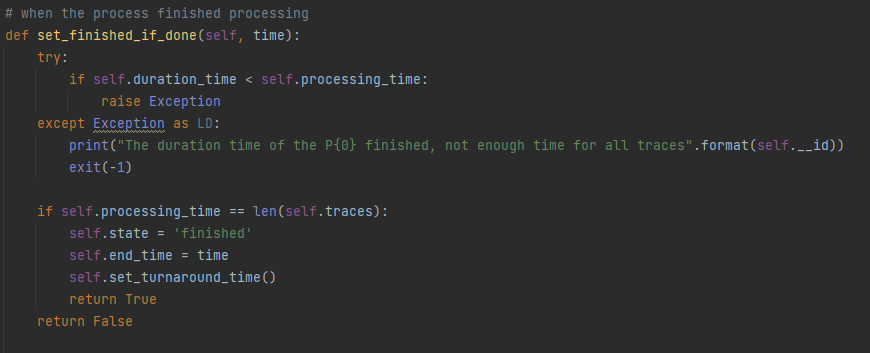


Figure 18: check if the process finish and change it state function

## Disk Class

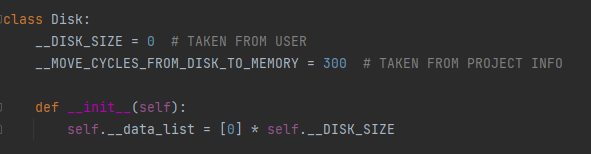


Figure 19: Disk Class

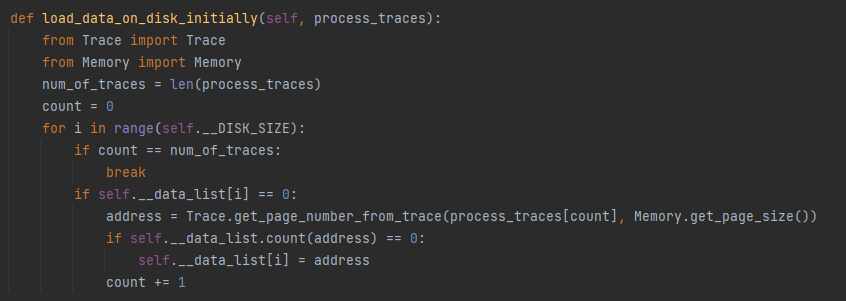


Figure 20: function for load data initially in the disk

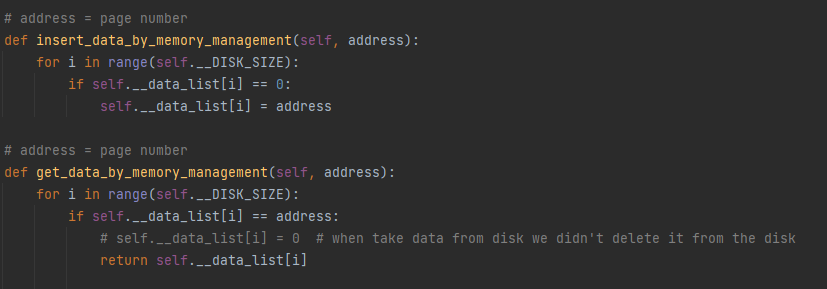


Figure 21: get and insert data in the disk functions

## Memory Class

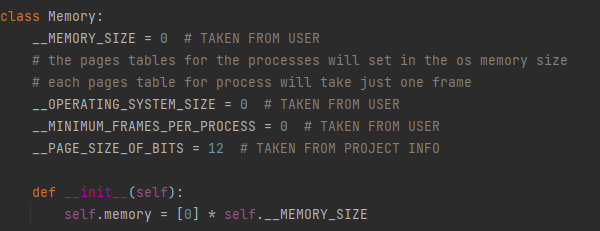


Figure 22: Memory Class

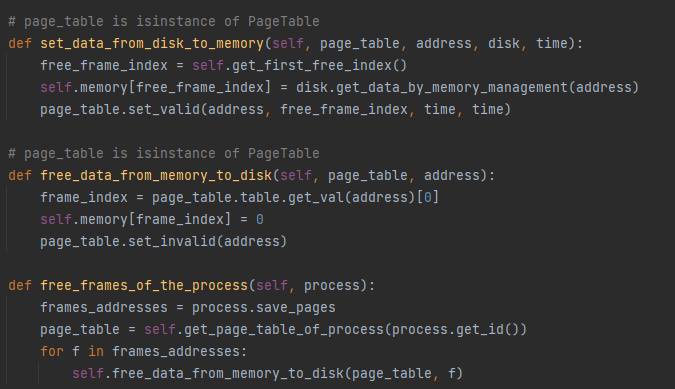


Figure 23: Set and free data from the memory functions

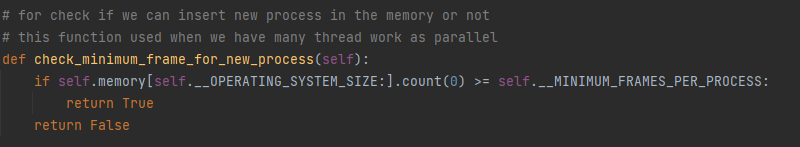


Figure 24: Check if there enough frames for new process function

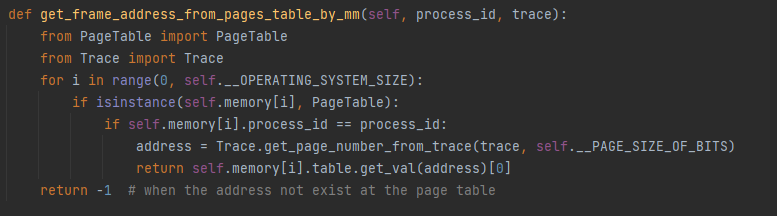


Figure 25: get the frame address for the process from the page table function

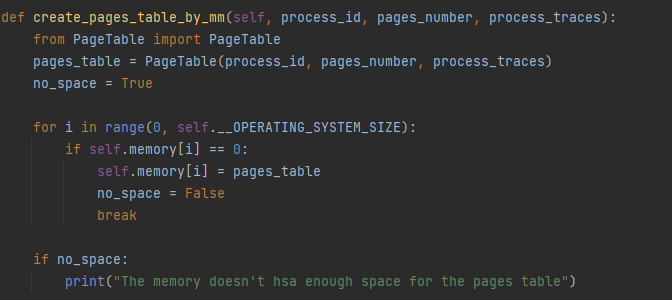


Figure 26: create pages table function

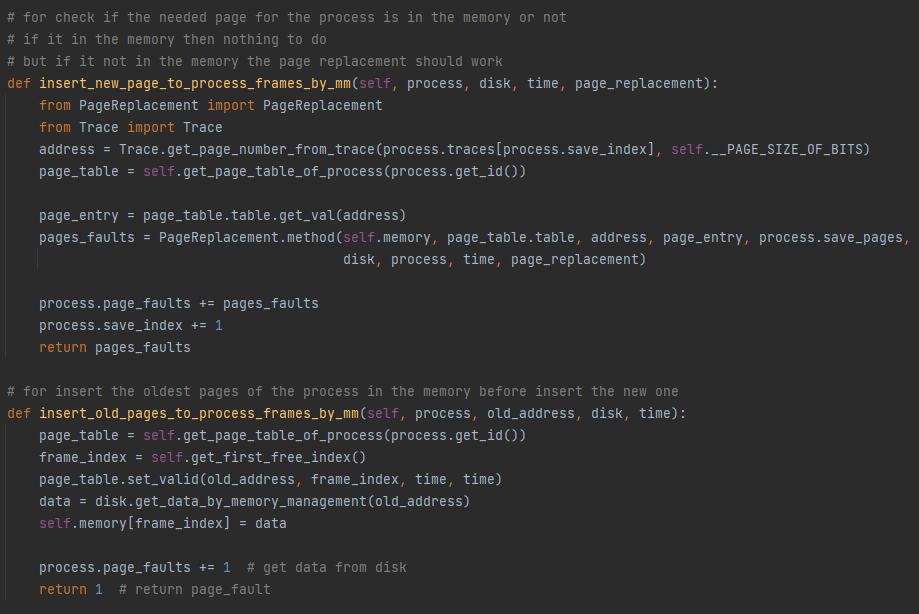


Figure 27: Insert new pages in the memory by use page replacement

## PageReplacment Class



Figure 28: Page Replacement Function

## Simulation Class

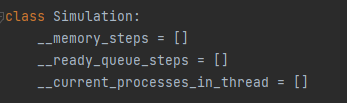


Figure 29: Simulation lists



Figure 30: add steps in the simulation lists functions

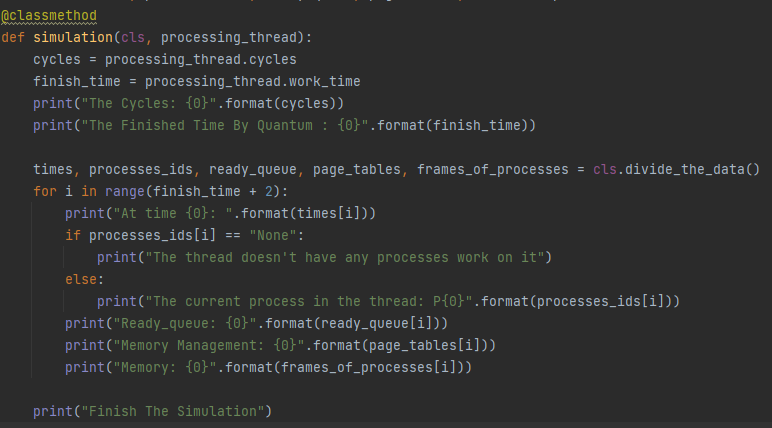


Figure 31: simulation function

## CPU Class

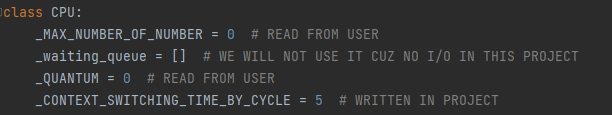


Figure 32: CPU Parameters

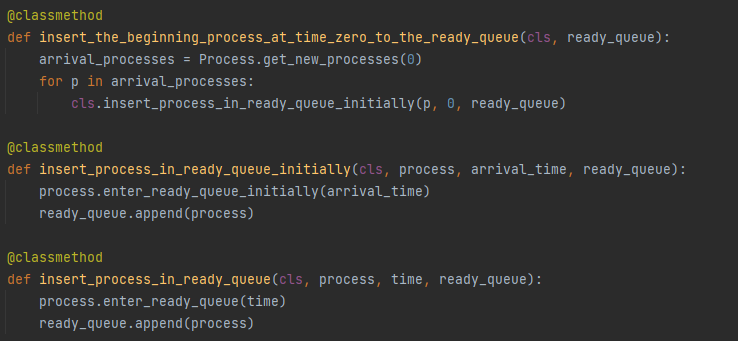


Figure 33: Insert process in the ready queue functions

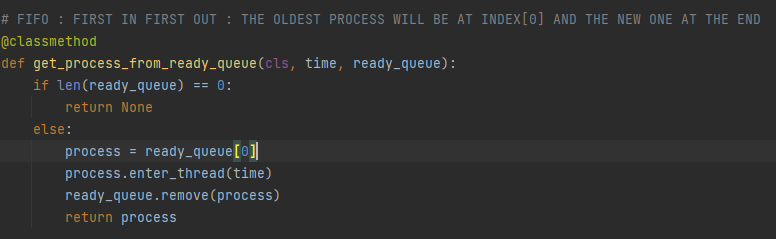


Figure 34: for get the process from the ready queue to insert it in the thread

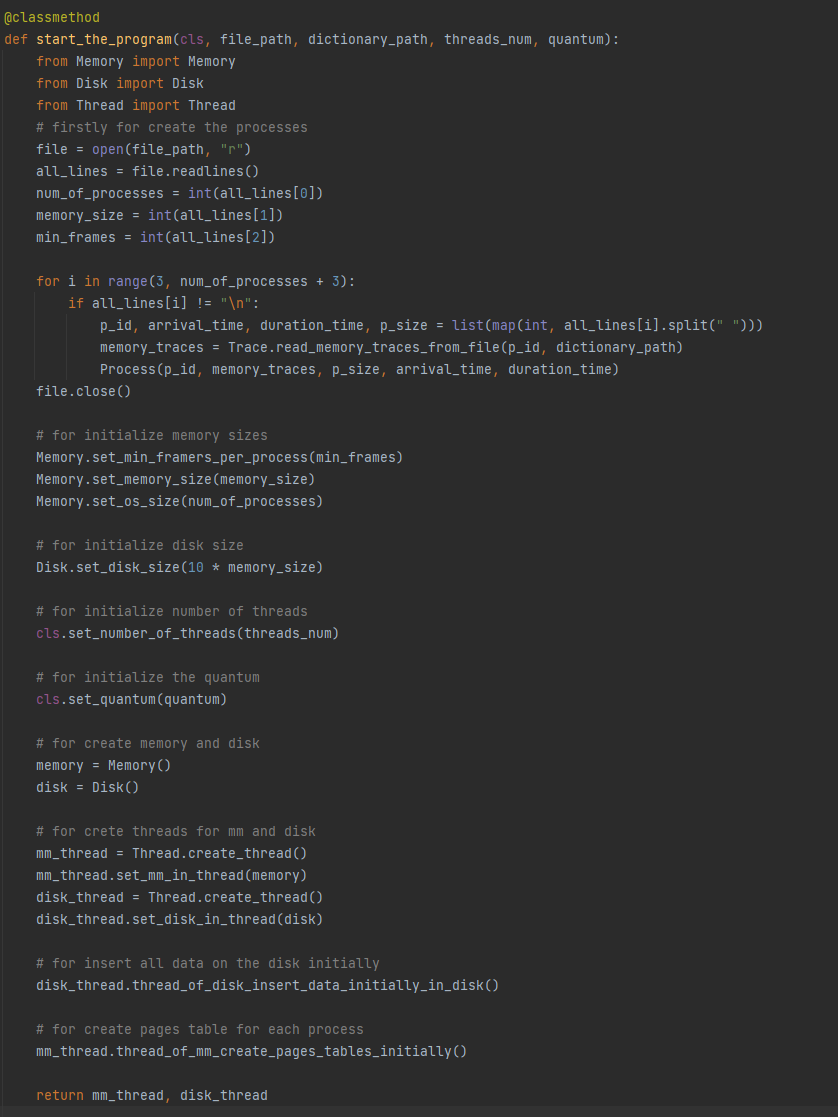


Figure 35: Start Program Function

## Thread Class

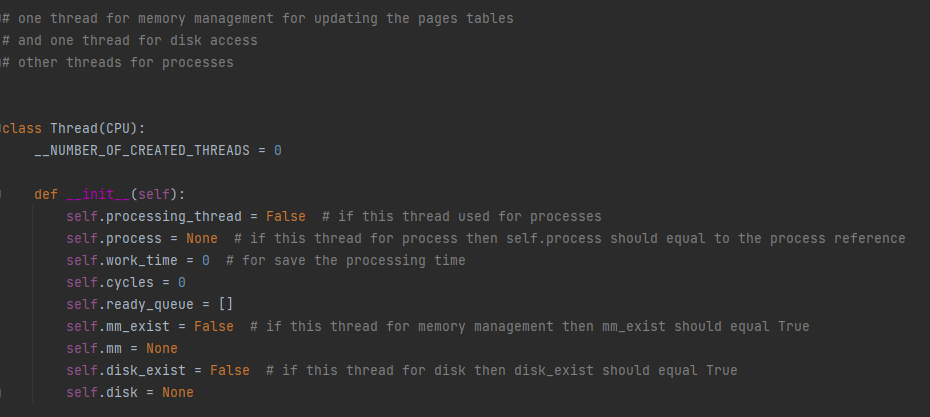


Figure 36: Thread Class

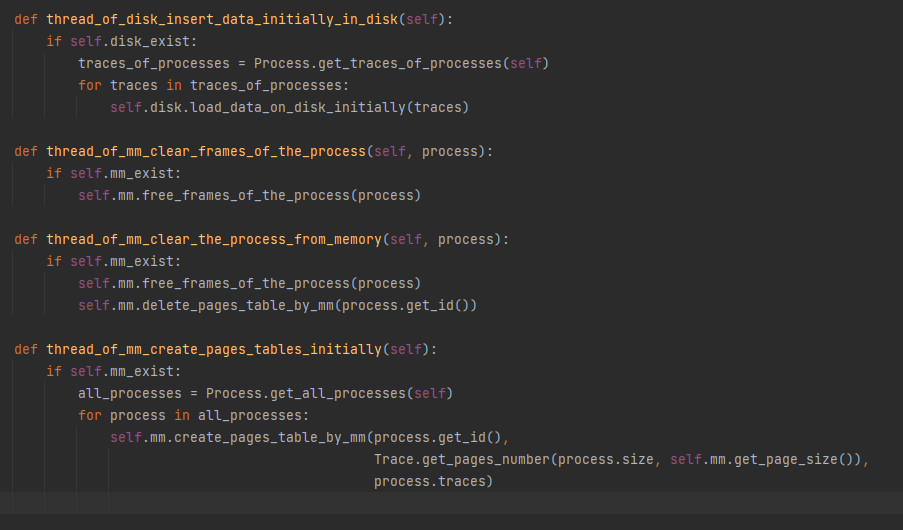


Figure 37: memory management and disk function by threads

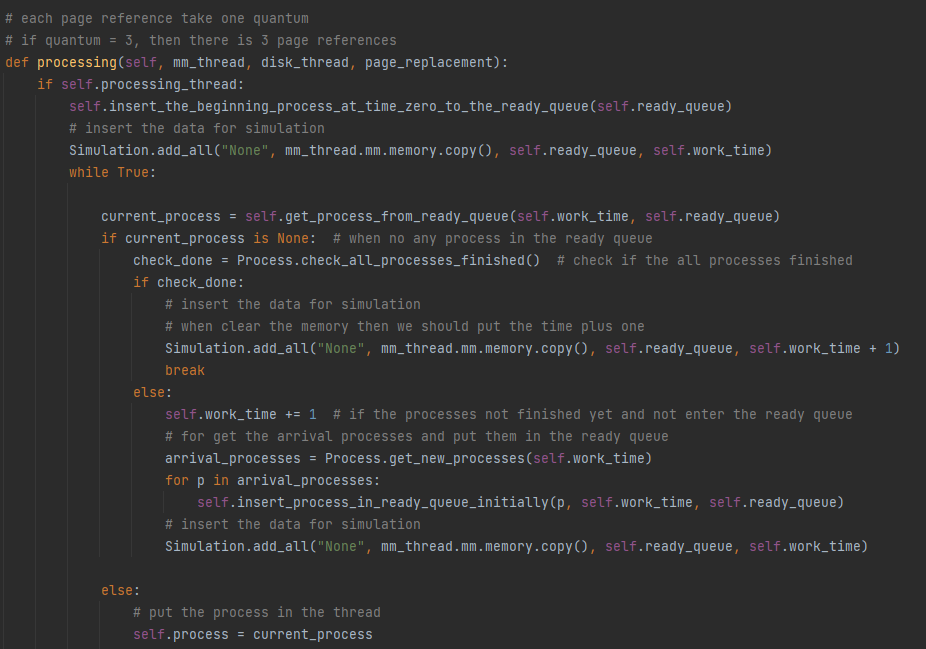


Figure 38: Processing Function of thread part 1

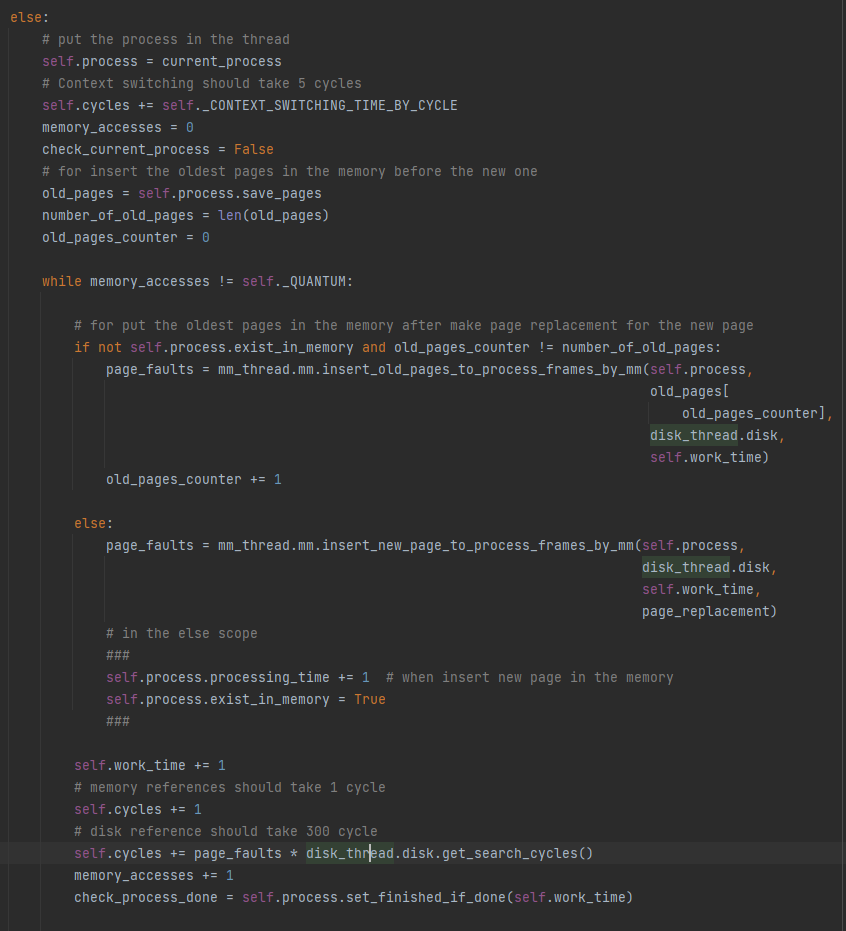


Figure 39: Processing Function of thread part 2

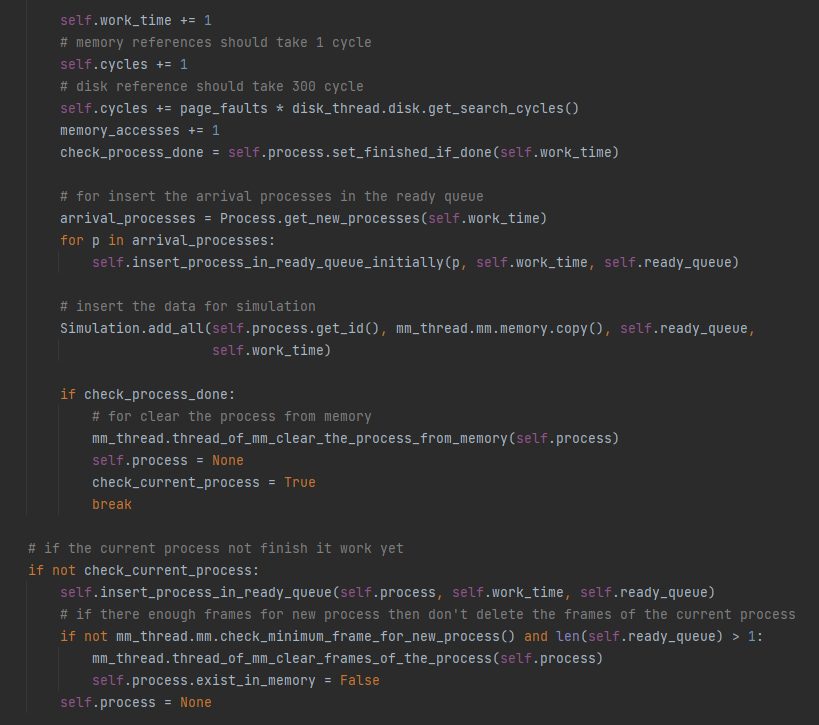
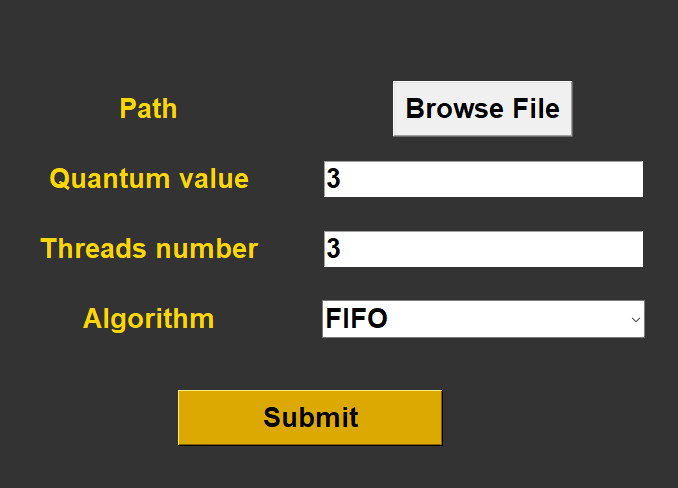


Figure 40: Processing Function of thread part 3

# Testing

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



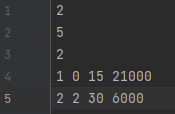
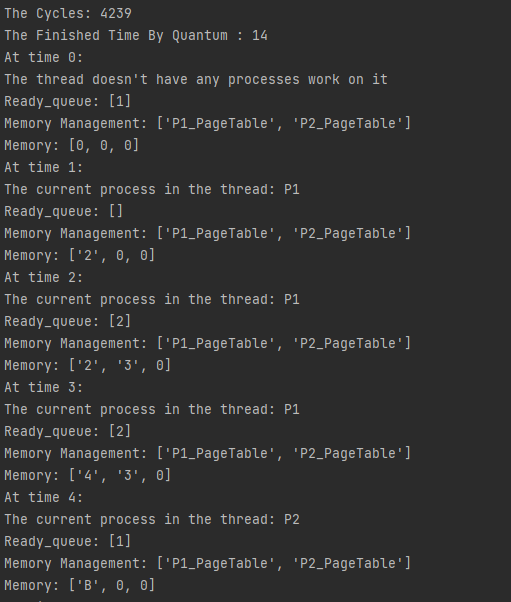
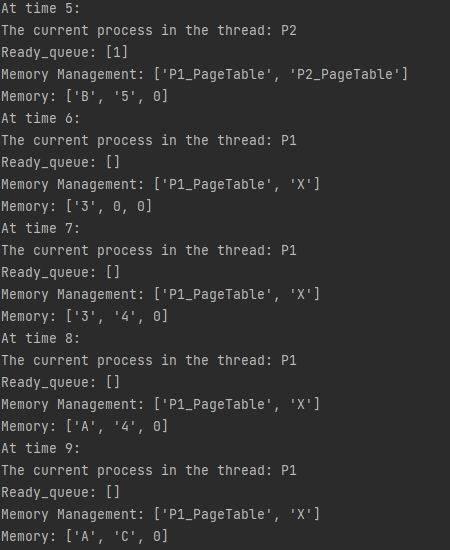
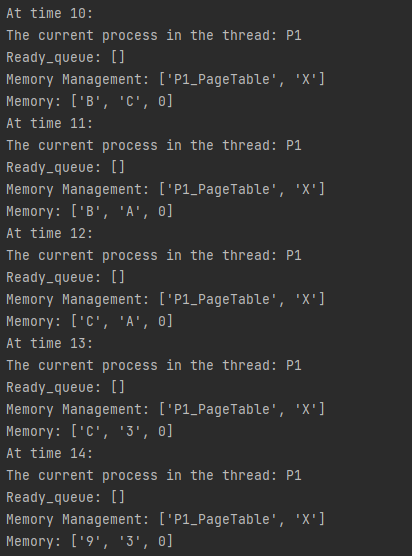


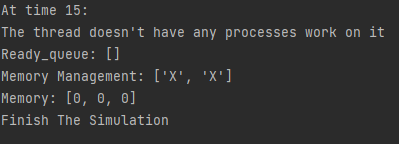
Figure 41: Exp1



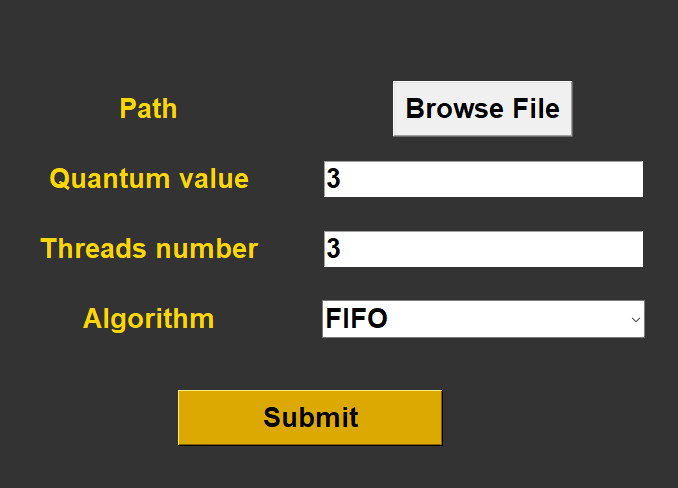








### Exp2: If there process in the memory and the new process will access by thread and want to enter the memory if there enough size then we shouldn’t delete the frames of the old process.



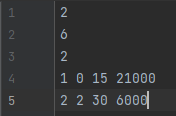
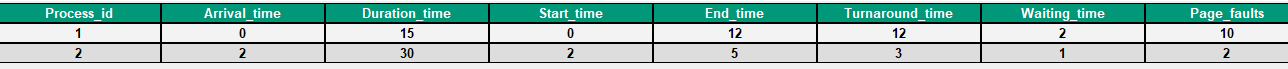
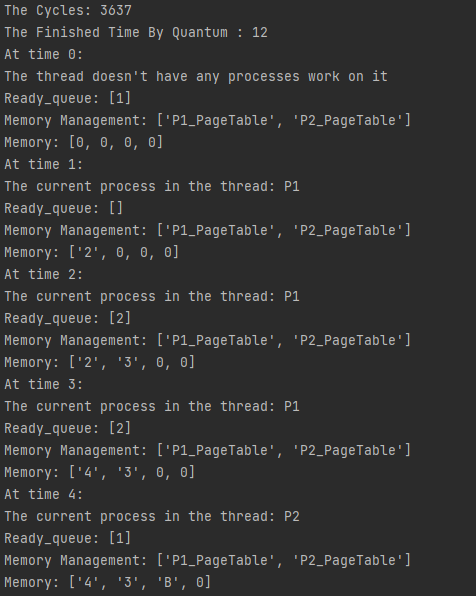
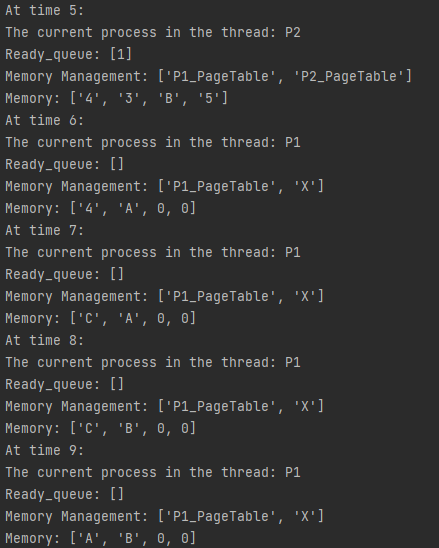
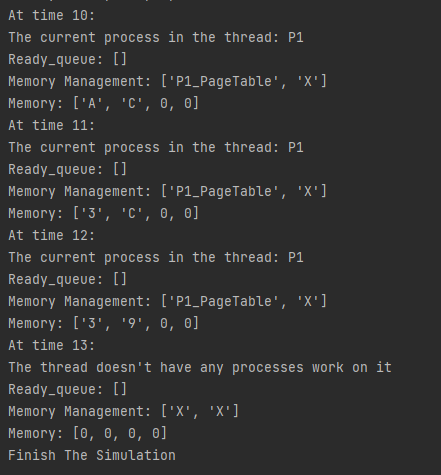


Figure 42: Exp2

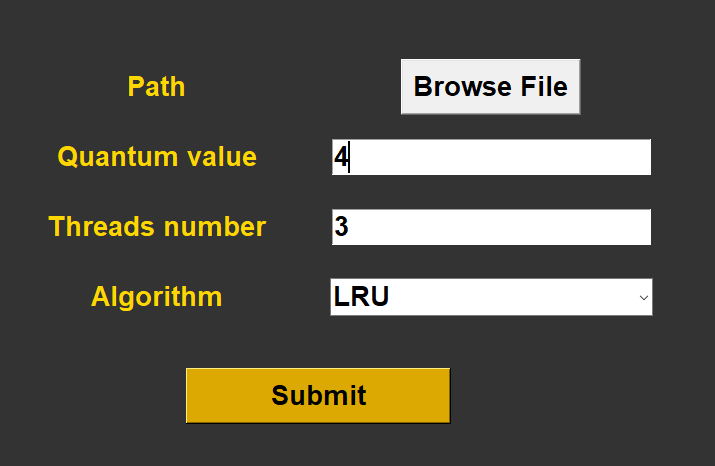








### Exp3: If there a process not arrive yet, then the thread will wait it until arrived, then the process go through thread and go in processing.



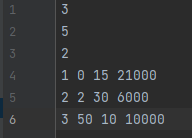
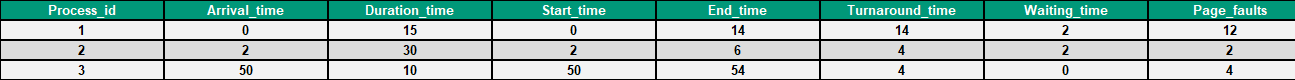
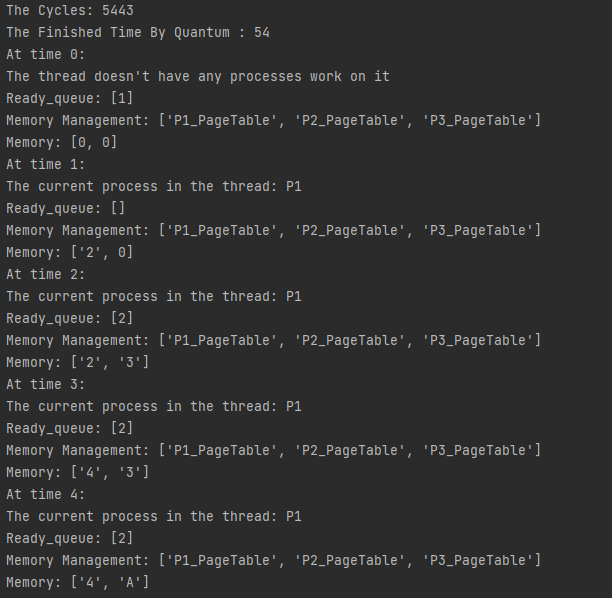
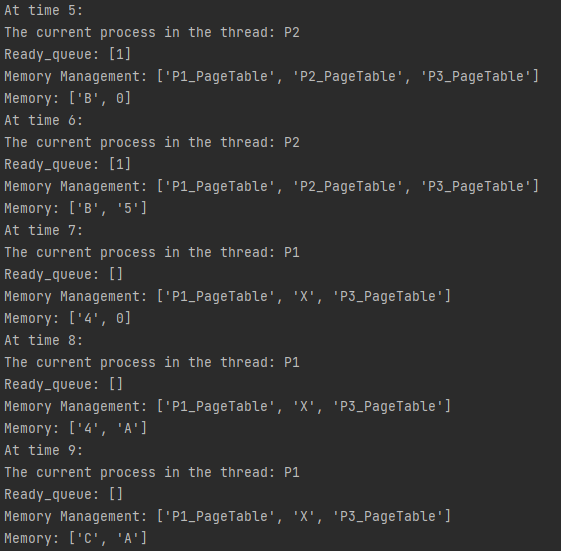
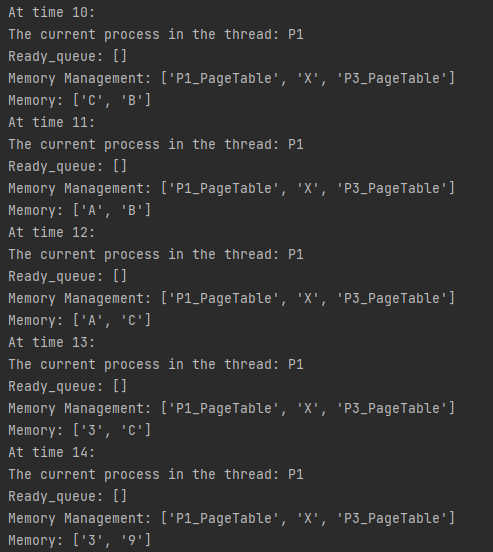


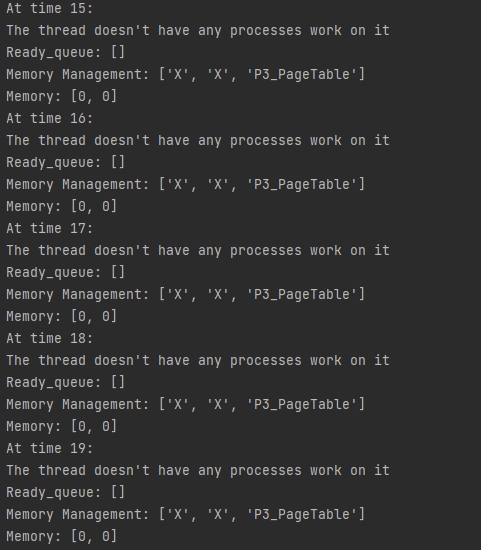
Figure 43: Exp3

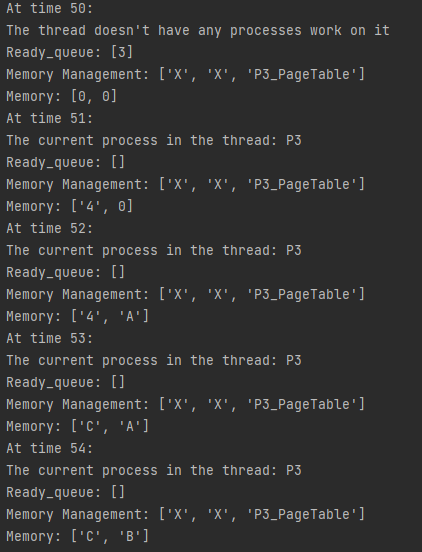












### Exp4: For see the different between the number of page fault when the data exist in the memory and request it again with the number of page fault when the data not exist in the memory and request it from the disk.

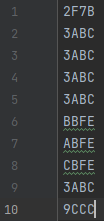


Figure 44: Exp4 part 1



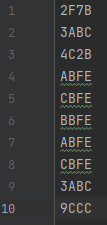


Figure 45: Exp4 part 2



# 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  
 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\_sizes\_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)  
  
 # 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()