**Name: Chen Yu Lin**

**Task 1**

Main memory size: 4MB

Page size: 4KB

|  |  |  |
| --- | --- | --- |
| Number of blocks | Number of pages | Size (KB) |
| 32 | 2 | 256 |
| 16 | 4 | 256 |
| 16 | 8 | 512 |
| 16 | 16 | 1024 |
| 16 | 32 | 2048 |

Algorithms

Memory Allocation: **Next Fit**, starts search from the block that is next on the list to the last allocation. Its’ advantage includes being faster than first fit and best fit, and avoids the overuse of memory at the start of the block chain but instead promotes a more even allocation of free memory.

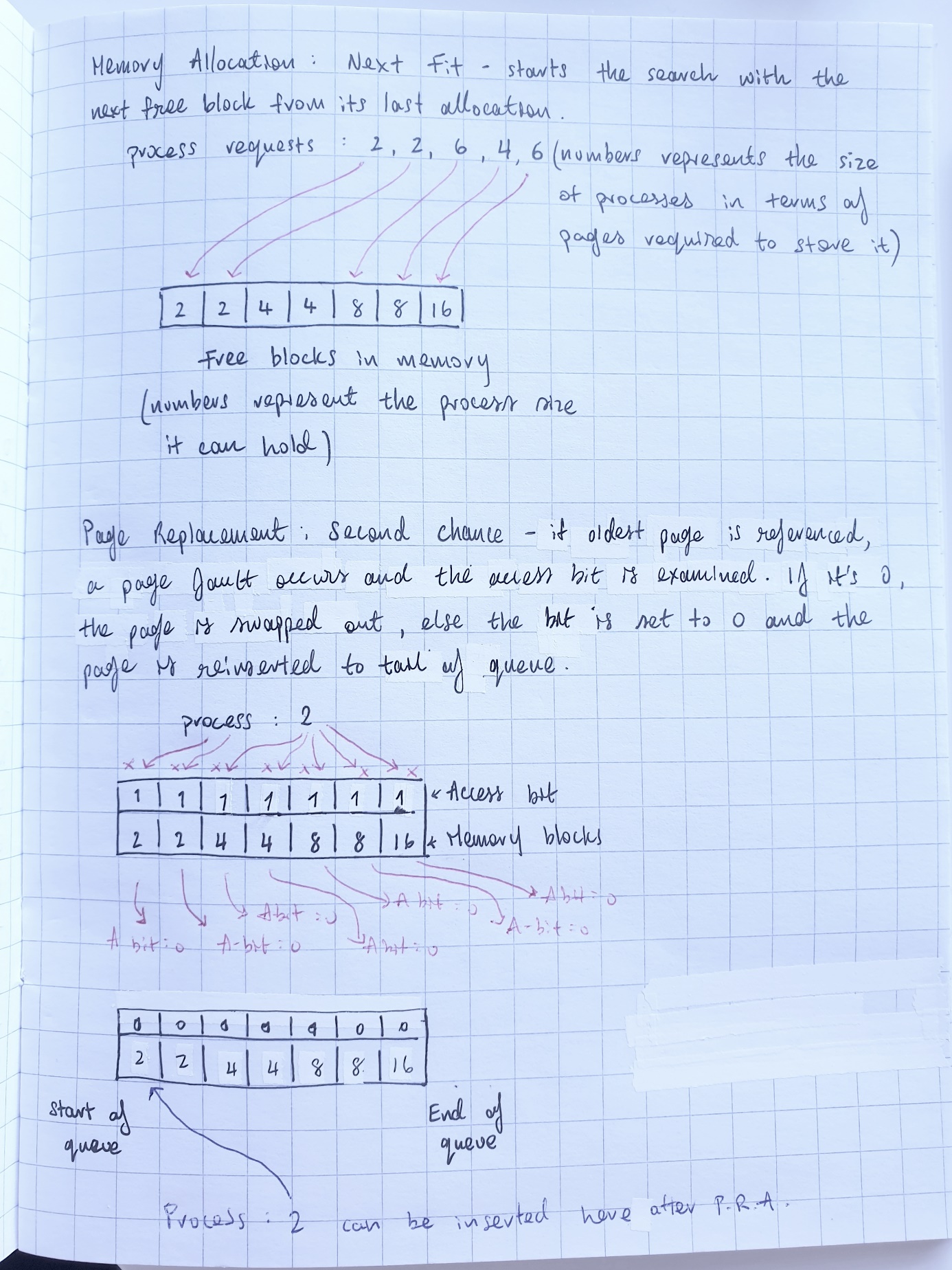
Page Replacement: **Second Chance**, this algorithm checks the access bit and if the page has not been used recently, it is swapped out. Otherwise, if the bit is 1, the bit will be cleared, and the page is inserted at the tail of the queue.

Data Structures

Memory block space: linked list

Memory requests: queue

Image on next page



**Task 2**

class Process:

# The Process Class – represents a process, each process has a process ID and a fixed size (in pages) #

p\_id = the process id

page\_size = the amount of pages required

class Block:

# The Block Class - may or may not contain a process #

process = the process object

pages = num of pages in the block

next = pointer for next block

prev = pointer for previous block

accessed = 0 or 1 # used to check when process was last accessed #

class Memory:

# Represents the memory space as a doubly linked list #

head = head of DLL

tail = tail of DLL

current = current node that is being pointed to

allocated\_list = [] # list of blocks that are currently occupied by a process #

def allocate(process):

if current == None:

current = head

while current != None:

if current.process == None and current.pages >= process.pages:

current.process = process

allocated\_list.append(current)

current = current.\_next

break

if current.\_next == None:

current = head

current = current.\_next

def page\_replace(process):

for index, block in enumerated(allocated\_list):

if block.pages >= process.pages :

if block.accessed == 0:

block.process = process

allocated\_list.append(allocated\_list.pop(index))

else:

block.accessed = 0

allocated\_list.append(allocated\_list.pop(index))

**Task 3**

I have also submitted the python file (just in case)

import random

# Data structures

# Memory Space : Doubly Linked List

# Memory Requests : queue

class Process:

    ''' Process Class - defines a process object, once a process object is instantiated, it is appended to the "requests" queue '''

    def \_\_init\_\_(self, pid, pages):

        self.\_pid = pid

        self.\_pages = 2\*\*pages

    # prints out a string representation of the process

    def \_\_str\_\_(self):

        return "process id: " + str(self.\_pid) + ", size: " + str(self.\_pages) + " pages"

class Block:

    ''' Block Class - defines the block object in the memory space (which has the structure of doubly-linked list).

        So each block is a node in the DLL. '''

    def \_\_init\_\_(self, pages):

        self.\_process = None        # data - process request

        self.\_previous = None       # prev pointer

        self.\_next = None           # next pointer

        self.\_pages = pages         # number of pages within the block

        self.\_accessed = 0          # data - access bit

    # prints out a string representation of the block

    def \_\_str\_\_(self):

        memory = "Block Info [" + "pages in block: " + str(self.\_pages) + " | access bit: " + str(self.\_accessed) + " | process details - " + str(self.\_process)  +  "]"

        return memory

class Memory:

    ''' Memory Class - defines the doubly-linked list object, which represents the "memory space" in this assignment '''

    requests = [] # a queue holding all incoming processes

    def \_\_init\_\_(self):

        self.\_head = None           # head of the DLL

        self.\_tail = None           # tail of the DLL

        self.\_current = None        # current node being pointed to

        self.\_allocated\_list = []   # the blocks currently occupied by a process

        self.\_size = 0              # number of blocks in the DLL

    # building the doubly-linked list object with the creation of blocks and their link pointers in the DLL

    def addBlock(self, data):

        newBlock = Block(data)

        self.\_size = self.\_size + 1

        if(self.\_head == None):

            self.\_head = self.\_tail = newBlock

            self.\_head.\_previous = None

            self.\_tail.\_next = None

        else:

            self.\_tail.\_next = newBlock

            newBlock.\_previous = self.\_tail

            self.\_tail = newBlock

            self.\_tail.\_next = None

    # returns the list that keeps track of the blocks that are currently occupied by a process

    def getAllocated(self):

        return self.\_allocated\_list

    # prints out a string representation of the memory space

    def \_\_str\_\_(self):

        used = {2:"2-U", 4:"4-U", 8:"8-U", 16:"16-U", 32:"32-U"}

        free = {2:"2-F", 4:"4-F", 8:"8-F", 16:"16-F", 32:"32-F"}

        current = self.\_head

        if(self.\_head == None):

            print("Error! Memory doesn't exist.")

            return

        print("Current Memory: \nPages \tAvailability")

        memory = ""

        memory += str(current.\_pages) + "\t"

        while(current != None):

            if current.\_process:

                memory += used[current.\_pages] + "(" + str(current.\_accessed) + ")" + " "

            else:

                memory += free[current.\_pages] + " "

            if current.\_next != None:

                if current.\_next.\_pages != current.\_pages:

                    memory += "\n" + str(current.\_next.\_pages) + "\t"

            current = current.\_next

        return memory

    # memory allocation is handled by the Next Fit algorithm

    def allocate(self, process):

        if self.\_current == None:

            self.\_current = self.\_head

        counter = 0

        while self.\_current != None:

            if self.\_current.\_process == None and self.\_current.\_pages >= process.\_pages:

                self.\_current.\_process = process

                self.\_allocated\_list.append(self.\_current)

                self.\_current = self.\_current.\_next

                return True

                break

            # when the end of the DLL is reached, reset current node to head

            # and traverse through list one more time to check for free blocks

            if self.\_current.\_next == None:

                self.\_current = self.\_head

                counter +=1

            # If there are still no free blocks even after the second full search,

            # then call the page replacement function

            if counter == 2:

                return self.page\_replace(process)

                counter = 0     # reset counter to 0

            self.\_current = self.\_current.\_next

    # page replacement is handled by the Second Chance algorithm

    def page\_replace(self, process):

        for index, block in enumerate(self.\_allocated\_list):

            if block.\_pages >= process.\_pages:

                if block.\_accessed == 0:        # if access bit is 0, then the page is swapped out and the new page is swapped in

                    print("\nPAGE REPLACING\n->", str(block), "->", process)

                    block.\_process = process

                    self.\_allocated\_list.append(self.\_allocated\_list.pop(index))

                    return True     # return True - page replacement happened successfully.

                else:

                    # if the access bit is 1, then it is set to 0 and the block is moved to the end of the list

                    print("Setting bit to 0 in this block --- ",str(block))

                    block.\_accessed = 0

                    self.\_allocated\_list.append(self.\_allocated\_list.pop(index))

        # return False - something went wrong, page could not be replaced and process was not inserted.

        return False

    # Manages the memory requests queue

    def memory\_request(self):

        if len(self.requests) == 0:

            return True

        process = self.requests[0]

        print("\nAllocating <<", str(process), ">> to memory")

        if len(self.requests) > 1:

            self.requests = self.requests[1::]

        else:

            self.requests = []

        success = self.allocate(process)

        if not success:

            print("ERROR! Process %s could not be allocated to memory!" % process)

    # randomly sets the access bits of some blocks to 1

    def accessed\_bit(self):

        block = self.\_allocated\_list[random.randint(0, len(self.\_allocated\_list)-1)]

        print(str(block), "has just been accessed! Setting its' access bit to 1")

        block.\_accessed = 1

    # randomly deallocates some pages as they have "finished" execution

    def deallocate(self):

        deallocate = random.randint(0, len(self.\_allocated\_list)-1)

        print("The process in the block << " + str(self.\_allocated\_list[deallocate]) + " >> has finished executing! Page will be deallocated now!")

        self.\_allocated\_list[deallocate].\_process = None

        self.\_allocated\_list.pop(deallocate)

# creating many blocks to be passed into the doubly-linked list (memory space)

def create\_block(num, size):

    for i in range(num):

        space.addBlock(size)

if \_\_name\_\_ == "\_\_main\_\_":

    space = Memory()

    create\_block(32, 2)

    create\_block(16, 4)

    create\_block(16, 8)

    create\_block(16, 16)

    create\_block(16, 32)

    num\_requests = 100

    # generating random processes to be added to the request queue

    print(num\_requests, "processes added to request queue...")

    for i in range(num\_requests):

        process = Process(random.randint(1000, 99999), random.randint(1, 5))

        space.requests.append(process)

    print("Memory Request Queue: ")

    for process in space.requests:

        print(str(process))

    # while there are processes in the request queue, allocate those processes to blocks in memory

    while True:

        finished = space.memory\_request()

        if finished:

            print("No more requests left...")

            print("\n\nMemory in it's final state\n", space)

            break

        list1 = space.getAllocated()

        print("In allocated list: ")

        for i in list1:

            print(i)

        print("\n", space, "\n", "-"\*30)

        if random.randint(0, 2) == 0:

            space.accessed\_bit()

        if random.randint(0, 3) == 0:

            space.deallocate()

**Task 4**

For the simulation of my code, I changed the number of blocks available in the memory space and lessened the number of requests to 20. The reason for this is that 100+ requests with 16-32 blocks per page of each size is way too long to copy and paste into this document.

Text

Description automatically generated

This is what the memory space would look like when there are some processes allocated to some blocks:

Text

Description automatically generated

U represents used

F represents full

The number within the bracket is the access bit.

So 2-U(0) means that : the block that has a size of 2 pages is occupied by a process, the access bit of this block is 0.

4-F means that : the block is free and empty, it doesn’t contain any blocks.

1. Processes are generated and appended to the Memory Requests Queue

A picture containing calendar

Description automatically generated

1. Processes from the request queue are allocated to blocks in the memory with Next Fit.

Text

Description automatically generated

1. Some processes may finish their execution, so they would be deallocated from the memory space.

Text

Description automatically generated

1. Sometimes some blocks may be accessed, so their access bit is set to 1.

Text

Description automatically generated

1. The algorithms are working as expected :

Text

Description automatically generated

Text

Description automatically generated with medium confidence

Text

Description automatically generated with medium confidence

Text

Description automatically generated

1. A page fault occurs because there are no free blocks in memory for a process! So the Second Chance algorithm handles the page replacement

Graphical user interface, text

Description automatically generated

Text

Description automatically generated

1. Showing the remainder of the output:

Graphical user interface, text

Description automatically generated

Graphical user interface, text

Description automatically generated

Graphical user interface, text

Description automatically generated

Graphical user interface, text

Description automatically generated

Text

Description automatically generated