

Lab 2 – Alexander Stradnic – 119377263

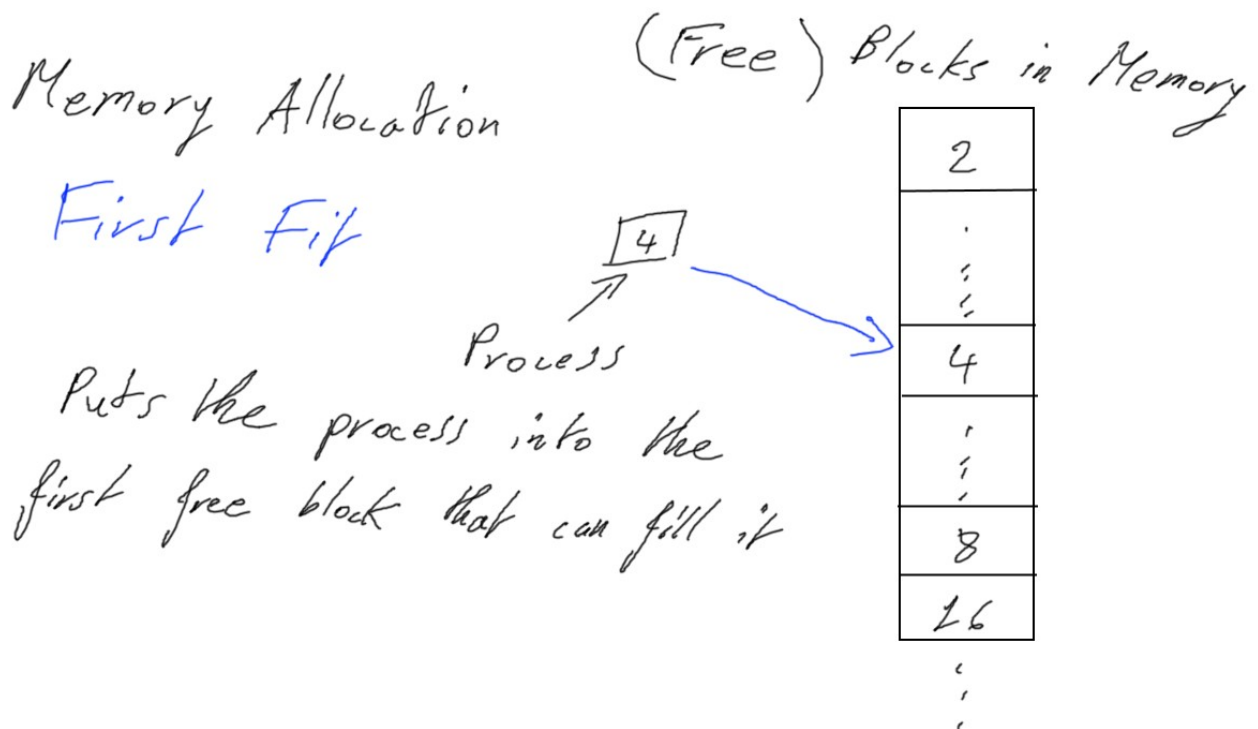
Task 1

Main memory size : 4MB

Page size : 4KB

No. of blocks	No. of Pages	Size (KB)
32	2	256
16	4	256
16	8	512
16	16	1024
16	32	2048

Memory Allocation : First Fit – this will use the first block that can fit the process requesting space



Page Replacement Algorithm : Second Chance – this will replace the oldest block if it hasn't been accessed recently

Page Replacement

Second Chance

the process is moved into the oldest block that can fit it if its accessed bit is 0. If it is 1 it is moved to the end of the queue and set to 0.

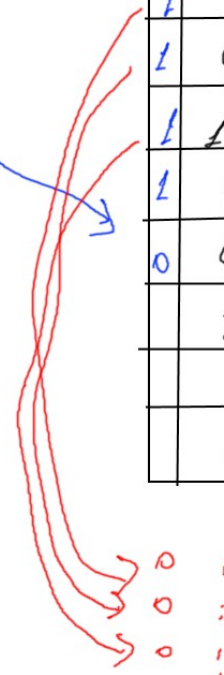
Execution

Queue

Oldest

0	2
1	8
1	4
1	16
1	4
0	4
	2
	.
	.

Newest



Space : Linked List

Memory requests : Queue

Task 2

requests := [List of Processes to be moved to main memory]

class Block:

#properties

pages := No of pages

next := Next block in Space
prev := Previous block in Space
process := The process itself
accessed := 1 if the process was accessed recently, else 0, used in the
Second Chance PRA

```
class Space:
    exec_list := A queue in which processes are sorted by their time of
    execution; the oldest being the first process
    first := Block(pages=2) # A DLL made up of blocks from 2 pages to
    32 pages
    function add(process):
        block = this.first
        while True:
            if not block.process and process.pages <= block.pages:
                block.process = process
                this.exec_list.append(block)
                break
            if not block.next:
                this.page_replace(process)

    function page_replace(process):
        for block in this.exec_list:
            if block.pages >= process and block.accessed == 0:
                block.process = process
                this.exec_list.append(this.exec_list.pop(block))
                return True
        return False

function process_requests():
    while True:
        process = requests[0]
        requests = requests[1::]
        success = space.page_replace(process)
        if not success:
            requests.append(process)
```

Task 3

```
import random
import time
```

```
# Dictionary of symbols representing empty and full blocks
full = {2:"■", 4:"■", 8:"▲", 16:"◆", 32:"●"}
empty = {2:"□", 4:"□", 8:"△", 16:"◇", 32:"○"}
```

```
class Process:
```

```
    """
```

```
    The Process Class - has a PID and takes up an amount of pages
```

```
    """
```

```
    pid = 0
```

```
    pages = 0
```

```
    def generate_pid(self, space, N=1000000):
```

```
        pid = random.randint(0, N-1)
```

```
        for process in space.requests:
```

```
            if process.pid == pid:
```

```
                return self.generate_pid(N)
```

```
        return pid
```

```
    def __init__(self, space, pages=-1):
```

```
        if pages == -1:
```

```
            pages = random.randint(0, 5)
```

```
            if pages == 0:
```

```
                pages = 1
```

```
            self.pages = 2**pages
```

```
            self.pid = self.generate_pid(space)
```

```
    def __str__(self):
```

```
        string = "PID: " + str(self.pid) + ", pages: " + str(self.pages)
```

```
        return string
```

```
class Block:
```

```
    """
```

The Block Class - Has an amount of pages, contains or doesn't contain a process,

has an accessed bit in line with Second Chance Page Replacement, and has pointers to the next and previous blocks in memory

'''

pages = 0

prev = None

next = None

process = None

accessed = 0

def __init__(self, pages, prev=None, next=None):

self.pages = pages

self.prev = prev

self.next = next

def __str__(self):

string = "Block: {a: " + str(self.accessed) + ", (" + (str(self.process) if self.process != None else "None") + "), prev pages: " + (str(self.prev.pages) if self.prev != None else "None") + ", next pages: " + (str(self.next.pages) if self.next != None else "None") + ", pages: " + str(self.pages) + "}"

return string

class Space:

'''

This represents the main memory of a machine and has a pointer to the first block,

a queue of incoming requests for memory,

and a list holding the order in which processes were allocated memory for page replacement

'''

exec_list = [] *# a queue maintaining the execution order of processes*

first = None

current = None

requests = [] *# a queue holding all incoming processes*

def __init__(self):

self.gen_blocks(32, 16, 16, 16, 16)

```
print(str(self.first))
```

```
def __str__(self):
```

```
    """
```

```
    Prints a visualisation of the memory in blocks
```

```
    """
```

```
    string = "-----\n"
```

```
    self.current = self.first
```

```
    string += str(self.current.pages) + "p\t"
```

```
    count = 0
```

```
    while self.current != None:
```

```
        count += 1
```

```
        if self.current.process:
```

```
            string += full[self.current.pages] + " "
```

```
        else:
```

```
            string += empty[self.current.pages] + " "
```

```
        if self.current.next != None:
```

```
            if self.current.next.pages != self.current.pages:
```

```
                string += "\n" + str(self.current.next.pages) + "p\t"
```

```
            elif count % 16 == 0:
```

```
                string += "\n\t"
```

```
            self.current = self.current.next
```

```
    string += "\n-----"
```

```
    # string += "\n"
```

```
    return string
```

```
def gen_blocks(self, num2, num4, num8, num16, num32):
```

```
    """
```

```
    Generates the memory blocks
```

```
    """
```

```
    self.first = Block(2)
```

```
    self.current = self.first
```

```
    self.gen_group(num2-1, 2)
```

```
    self.gen_group(num4, 4)
```

```
    self.gen_group(num8, 8)
```

```
    self.gen_group(num16, 16)
```

```
    self.gen_group(num32, 32)
```

```
def gen_group(self, num, size):
```

```
    """
```

```
        More specific method that handles generating each group of blocks  
(grouped by size)
```

```
    """
```

```
    for i in range(num):
```

```
        block = Block(size, self.current)
```

```
        self.current.next = block
```

```
        self.current = block
```

```
def add(self, process):
```

```
    """
```

```
        This method adds a process to a memory block using First Fit  
principles, otherwise calls page_replace
```

```
    """
```

```
    block = self.first
```

```
    while True:
```

```
        if block.process == None and process.pages <= block.pages:
```

```
            block.process = process
```

```
            self.exec_list.append(block)
```

```
            return True
```

```
        elif block.next == None:
```

```
            return self.page_replace(process)
```

```
        block = block.next
```

```
def page_replace(self, process):
```

```
    """
```

```
        Handles page replacement using the Second Chance PRA
```

```
    """
```

```
    for i, block in enumerate(self.exec_list):
```

```
        if block.pages >= process.pages:
```

```
            if block.accessed == 0: # if the block has not been accessed  
recently, then replace the process and add to the end of the exec_list
```

```
                print("Replacing process in", str(block), " with", process)
```

```
                block.process = process
```

```
                self.exec_list.append(self.exec_list.pop(i))
```

```
                # print([str(block) for block in self.exec_list])
```

```
                return True
```

```

        else: # else set the accessed bit to 0, and move to the end of the
exec_list
        block.accessed = 0
        self.exec_list.append(self.exec_list.pop(i))
    return False

def gen_requests(self, n=20):
    """
    Generates memory requests
    """
    print("Generating", n, "processes requesting space...")
    for i in range(n):
        self.requests.append(Process(self))
    print("Handling requests: ", [str(process) for process in self.requests])

def process_request(self):
    """
    Handles process requests to memory, if it was not able to be added,
    then it is moved to the back of the requests queue
    """
    if len(self.requests) == 0:
        return True
    process = self.requests[0]
    print("Handling process", str(process))
    if len(self.requests) > 1:
        self.requests = self.requests[1:]
    else:
        self.requests = []
    success = self.add(process)
    time.sleep(0.5)
    if not success:
        self.requests.append(process)
    print(str(self))
    self.frag()

def frag(self):
    """
    Prints out the amount of blocks and pages affected by internal and
    external fragmentation

```



```

'''
self.current = self.first
external_frag = 0
external_blocks = 0
internal_frag = 0
internal_blocks = 0
while self.current != None:
    if self.current.process == None:
        external_frag += self.current.pages
        external_blocks += 1
    else:
        if self.current.process.pages < self.current.pages:
            internal_frag += self.current.pages - self.current.process.pages
            internal_blocks += 1
        self.current = self.current.next
print("Internal fragmentation:", internal_frag, "pages,",
internal_blocks, "blocks affected\nExternal fragmentation:", external_frag,
"pages,", external_blocks, "empty block(s)\n")

def set_used(self):
'''
A method which randomly selects blocks to set their accessed bit to 0
or 1, simulating recent access of those memory blocks
'''
amount = random.randint(0, len(self.exec_list) // 4)
while amount > 0:
    block = self.exec_list[random.randint(0, len(self.exec_list)-1)]
    if block.accessed == 1:
        block.accessed = 0
    else:
        block.accessed = 1
    amount -= 1

space = Space()
print(str(space))
space.gen_requests(100)
time.sleep(5)
while True:
    no_reqs = space.process_request()

```

```
if no_reqs:
    print("No more requests left...")
    break
space.set_used()
```

Task 4

The program starts by creating a Space, simulating the main memory consisting of blocks. It runs an infinite loop during which processes are added (and replaced when needed) from memory. The execution of processes is not simulated, only their allocation and replacement.

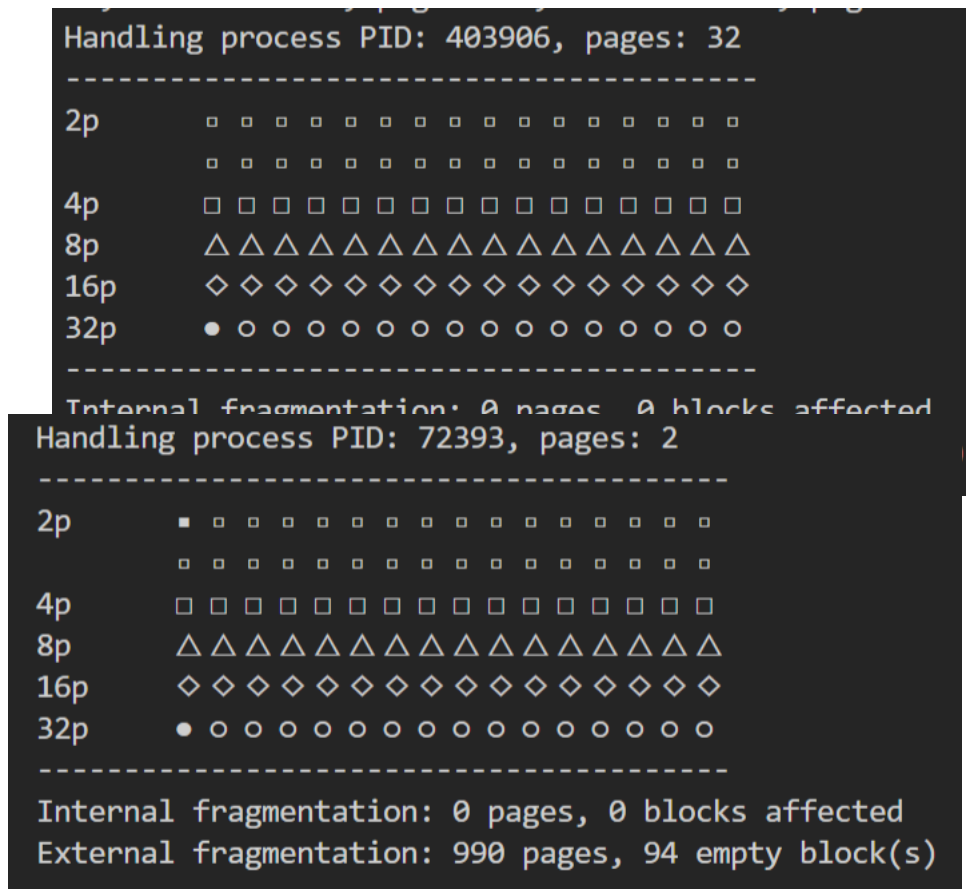
First the blocks are created, with space.first pointing to the first block



Then the processes are generated and added to a queue of memory requests

```
Generating 100 processes requesting space...
Handling requests: ['PID: 403906, pages: 32', 'PID: 72393, pages: 2', 'PID: 732145, pages: 2', 'PID: 628214, pages: 2', 'PID: 676978, pages: 8', 'PID: 898772, pages: 16', 'PID: 941765, pages: 16', 'PID: 724256, pages: 16', 'PID: 364479, pages: 16', 'PID: 366525, pages: 2', 'PID: 724534, pages: 32', 'PID: 74508, pages: 2', 'PID: 253985, pages: 8', 'PID: 253223, pages: 16', 'PID: 408417, pages: 4', 'PID: 521856, pages: 8', 'PID: 805655, pages: 2', 'PID: 257335, pages: 2', 'PID: 107312, pages: 4', 'PID: 741269, pages: 2', 'PID: 907033, pages: 4', 'PID: 639296, pages: 4', 'PID: 651018, pages: 16', 'PID: 802844, pages: 2', 'PID: 103140, pages: 2', 'PID: 808463, pages: 8', 'PID: 102040, pages: 32', 'PID: 385363, pages: 4', 'PID: 934340, pages: 2', 'PID: 841580, pages: 4', 'PID: 418940, pages: 4', 'PID: 374392, pages: 4', 'PID: 870979, pages: 8', 'PID: 149245, pages: 16', 'PID: 274235, pages: 2', 'PID: 599426, pages: 2', 'PID: 475775, pages: 8', 'PID: 657020, pages: 2', 'PID: 185930, pages: 16', 'PID: 661842, pages: 16', 'PID: 18571, pages: 2', 'PID: 544272, pages: 16', 'PID: 963622, pages: 32', 'PID: 324383, pages: 16', 'PID: 510788, pages: 16', 'PID: 734744, pages: 4', 'PID: 379183, pages: 2', 'PID: 426628, pages: 8', 'PID: 914873, pages: 16', 'PID: 603887, pages: 2', 'PID: 675592, pages: 16', 'PID: 727596, pages: 8', 'PID: 829913, pages: 2', 'PID: 621638, pages: 4', 'PID: 443058, pages: 4', 'PID: 650216, pages: 32', 'PID: 340539, pages: 32', 'PID: 978254, pages: 8', 'PID: 838080, pages: 16', 'PID: 483795, pages: 2', 'PID: 69675, pages: 32', 'PID: 323145, pages: 32', 'PID: 651302, pages: 2', 'PID: 376426, pages: 16', 'PID: 183832, pages: 16', 'PID: 928545, pages: 2', 'PID: 833303, pages: 4', 'PID: 285275, pages: 2', 'PID: 659284, pages: 16', 'PID: 163569, pages: 2', 'PID: 367610, pages: 4', 'PID: 717440, pages: 8', 'PID: 670736, pages: 2', 'PID: 874253, pages: 16', 'PID: 182207, pages: 2', 'PID: 783947, pages: 2', 'PID: 783339, pages: 4', 'PID: 174330, pages: 16', 'PID: 747864, pages: 8', 'PID: 727428, pages: 8', 'PID: 38350, pages: 8', 'PID: 554677, pages: 8', 'PID: 180479, pages: 4', 'PID: 712583, pages: 4', 'PID: 418878, pages: 16', 'PID: 404048, pages: 2', 'PID: 192198, pages: 4', 'PID: 72627, pages: 2', 'PID: 688839, pages: 16', 'PID: 140476, pages: 32', 'PID: 571622, pages: 2', 'PID: 811888, pages: 2', 'PID: 677557, pages: 16', 'PID: 957339, pages: 16', 'PID: 183314, pages: 2', 'PID: 619911, pages: 2', 'PID: 922910, pages: 16', 'PID: 531593, pages: 8', 'PID: 903394, pages: 16', 'PID: 179777, pages: 2']
Handling process PID: 403906, pages: 32
```

Each process is then handled in the infinite loop – initially as all blocks are empty they are all added with no replacement. The internal and external fragmentation of Space is reported after each iteration



As the memory is filled, a process has to be replaced. This is done using the Second Chance PRA, and in my implementation random blocks' accessed bit is changed to simulate usage

```
Handling process PID: 957339, pages: 16
Replacing process in Block: {a: 0, (PID: 898772, pages: 16), prev pages: 8, next pages: 16, pages: 16} with PID: 957339, pages: 16
-----
2p  . . . . .
4p  . . . . .
8p  . . . . .
16p . . . . .
32p . . . . .
-----
Internal fragmentation: 116 pages, 8 blocks affected
External fragmentation: 12 pages, 3 empty block(s)
```

This continues until the requests queue is empty

```
Handling process PID: 179777, pages: 2
Replacing process in Block: {a: 0, (PID: 72393, pages: 2), prev pages: None, next pages: 2, pages: 2} with PID: 179777, pages: 2
-----
2p  . . . . .
4p  . . . . .
8p  . . . . .
16p . . . . .
32p . . . . .
-----
Internal fragmentation: 116 pages, 8 blocks affected
External fragmentation: 0 pages, 0 empty block(s)

No more requests left...
```