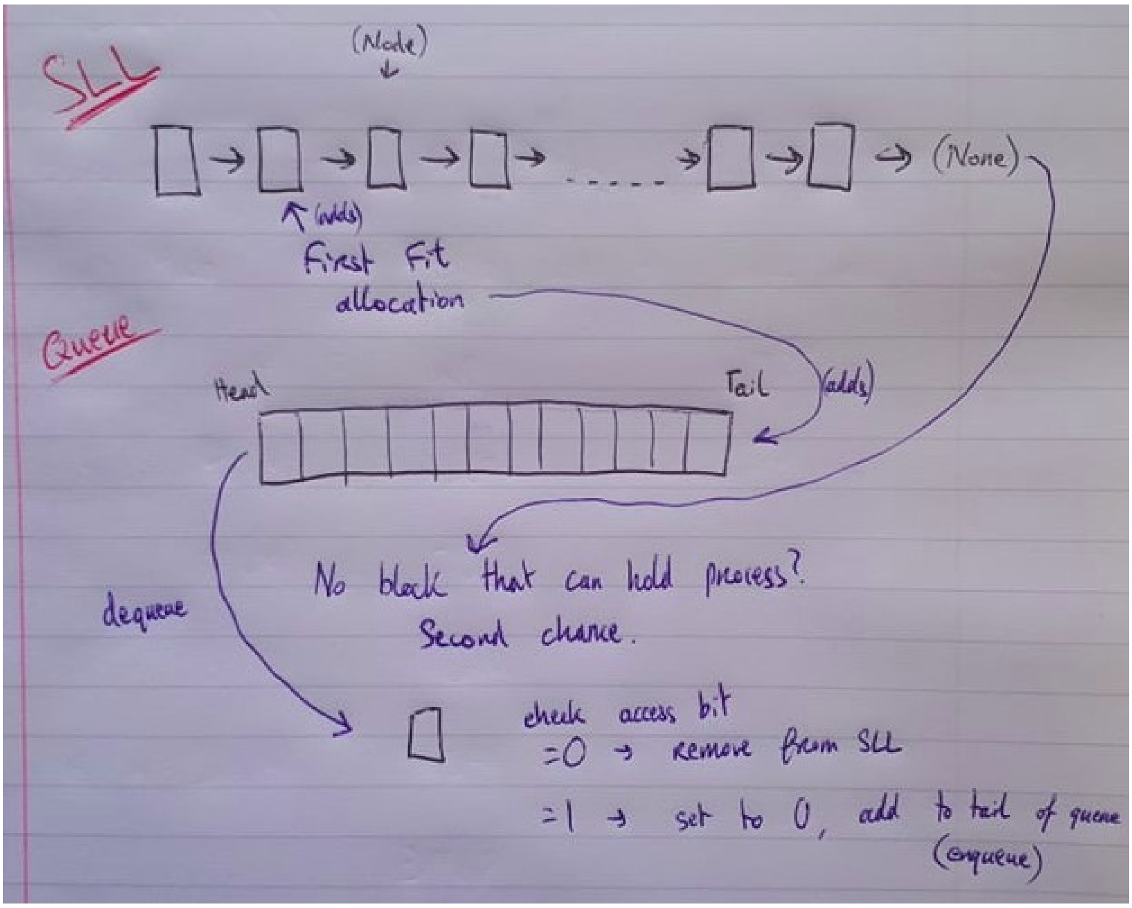
Lab: 2
Name: Peadar O’Connor
Student Number: 117302273

Task 1:
Main memory = 4 MB (4096kb)
Page size = 4 KB
Organization in blocks: 48 blocks of 2 pages, 32 blocks of 4 pages, 20 blocks of 8 pages, 16 blocks of 16 pages, 12 blocks of 32 pages.

Linked list for free memory
Memory allocation algorithm: First fit
Why? - My organization of blocks has more blocks in the low memory area, which accommodates for first fit’s tendency to have allocations towards the low memory addresses.
Data structure? - Linked list

Page replacement algorithm: Second chance
Why? - Improves on FIFO, lets pages that are still being used to not be replaced.
Data structure? - Queue

The two algorithms interact when the first fit algorithm adds something to memory. This page is added to the queue within the second chance algorithm, creating the queue on which the replacement will occur.
They also interact when the first fit algorithm sees that there is no more blocks left that fit the process, and then calls for the page replacement algorithm to be used to free up some pages.



Task 2:

First fit algorithm:

process, linked list, queue is provided
added = False
for each block in the linked list starting from the first:
    if the block's remaining space is large enough to fit the process:
        process assigned to block
        processes used_pages = process size / page size (rounded up math.ceil())
        block's remaining pages -= processes used_pages
        block's remaining space = block's remaining pages * page size
        added = True
        add to second chance queue
        break
if added = False:
    if theres nothing in the queue:
        Error: Process too big
    else:
        call on the page replacement algorithm
        try to add the process again

page de-allocation method:

process, linked list provided
for each block in the linked list starting from the first:
    if process to be removed is in this block:
        blocks remaining pages += processes used_pages
        block's remaining space = block's remaining pages * page size
        set processes access bit to 0 in second chance queue
        process removed from block

Assumption: page de-allocation can happen at any time, as well as being called by the second chance queue to replace pages (said in the lab sheet: During the execution, some pages are de-allocated and become free again – the explanation is that processes finished the execution and their pages were released.) Because of this the access bit is set to 0 so that if the page is de-allocated at any time, the second chance algorithm will know that the process is done with.

Second chance algorithm:

remove process from head of the queue
if accessed bit of process = 0:
    call page de-allocation method on the process
else:
    process access bit = 0
    add to tail of second chance queue

Task 3:

```
# Peadar O'Connor 117302273
import math


class SLLNode:
    """
    A node for use in the Singly linked list ADT.

    Can hold an element and the next node in the list.
    """
    def __init__(self, item, nextnode):
        self.element = item     #any object
        self.next = nextnode    #an SLLNode


class SLinkedList:
    """
    A Singly linked list made up of linked nodes.

    Each node is linked to the next one in the list in order.

    This SLL ADT is taken from CS2515:Algorithms and Data Structures I
    """
    def __init__(self):
        self.first = None
        self.size = 0

    def add_first(self, element):
        """ Add node to the start of the list """
        node = SLLNode(element, self.first)
        self.first = node
        self.size = self.size + 1

    def get_first(self):
        """ Return the first element of the list """
        if self.size == 0:
            return None
        return self.first.element

    def get_first_node(self):
        """ Return the first node of the list """
        if self.size == 0:
            return None
        return self.first

    def remove_first(self):
        """ Remove the first element of the list """
        if self.size == 0:
            return None
        item = self.first.element
        self.first = self.first.next
        self.size = self.size - 1
        return item

    def get_length(self):
        """ Return the length of the list. """
        length = 0
        current = self.get_first()
        while current:
            current = current.next
            length += 1
        return length


class Queue:
    """
    A queue using a python list, with internal wrap-around..

    Head and tail of the queue are maintained by internal pointers.
    When the list is full, a new bigger list is created.

    This queue ADT is taken from CS2515:Algorithms and Data Structures I
    """

    def __init__(self):
```

```python
        self.body = [None] * 10
        self.head = 0  # index of first element, but 0 if empty
        self.tail = 0  # index of free cell for next element
        self.size = 0  # number of elements in the queue

    def __str__(self):
        output = '<-'
        i = self.head
        if self.head < self.tail:
            while i < self.tail:
                output = output + str(self.body[i]) + '-'
                i = i + 1
        else:
            while i < len(self.body):
                output = output + str(self.body[i]) + '-'
                i = i + 1
            i = 0
            while i < self.tail:
                output = output + str(self.body[i]) + '-'
                i = i + 1
        output = output + '<'
        output = output + '    ' + self.summary()
        return output

    def get_size(self):
        """ Return the internal size of the queue. """
        return sys.getsizeof(self.body)

    def summary(self):
        """ Return a string summary of the queue. """
        return ('Head:' + str(self.head)
                + '; tail:' + str(self.tail)
                + '; size:' + str(self.size))

    def grow(self):
        """ Grow the internal representation of the queue.

        This should not be called externally.
        """
        # print('growing')
        # print('Before growing:')
        # print(self)
        oldbody = self.body
        self.body = [None] * (2 * self.size)
        oldpos = self.head
        pos = 0
        if self.head < self.tail:  # data is not wrapped around in list
            while oldpos <= self.tail:
                self.body[pos] = oldbody[oldpos]
                oldbody[oldpos] = None
                pos = pos + 1
                oldpos = oldpos + 1
        else:  # data is wrapped around
            while oldpos < len(oldbody):
                self.body[pos] = oldbody[oldpos]
                oldbody[oldpos] = None
                pos = pos + 1
                oldpos = oldpos + 1
            oldpos = 0
            while oldpos <= self.tail:
                self.body[pos] = oldbody[oldpos]
                oldbody[oldpos] = None
                pos = pos + 1
                oldpos = oldpos + 1
        self.head = 0
        self.tail = self.size

    def enqueue(self, item):
        """ Add an item to the queue.

        Args:
            item - (any type) to be added to the queue.
        """
        # An improved representation would use modular arithmetic
        if self.size == 0:
            self.body[0] = item  # assumes an empty queue has head at 0
            self.size = 1
            self.tail = 1
```

```python
        else:
            self.body[self.tail] = item
            # print('self.tail =', self.tail, ': ', self.body[self.tail])
            self.size = self.size + 1
            if self.size == len(self.body):  # list is now full
                self.grow()  # so grow it ready for next enqueue
            elif self.tail == len(self.body) - 1:  # no room at end, but must be at front
                self.tail = 0
            else:
                self.tail = self.tail + 1
        # print(self)

    def dequeue(self):
        """ Return (and remove) the item in the queue for longest. """
        # An improved implementation would use modular arithmetic
        if self.size == 0:  # empty queue
            return None
        item = self.body[self.head]
        self.body[self.head] = None
        if self.size == 1:  # just removed last element, so rebalance
            self.head = 0
            self.tail = 0
            self.size = 0
        elif self.head == len(self.body) - 1:  # if head was the end of the list
            self.head = 0  # we must have wrapped round, so point to start
            self.size = self.size - 1
        else:
            self.head = self.head + 1  # just move the pointer on one cell
            self.size = self.size - 1
        # we haven't changed the tail, so nothing to do
        return item

    def length(self):
        """ Return the number of items in the queue. """
        return self.size

    def first(self):
        """ Return the first item in the queue. """
        return self.body[self.head]  # will return None if queue is empty


class Process:
    """
    A class for a memory-using process. It has an id and a set size
    for use in the memory allocation algorithm.
    """
    def __init__(self, id, size):
        self.id = id
        self.size = size
        self.access_bit = 1
        self.used_pages = None


class Block:
    """
    A class for a block in the memory. Each block has a number of pages
    assigned, and how big these pages are determine the total space allocated
    to the block.
    """
    def __init__(self, max_pages, page_size):
        self.max_pages = max_pages
        self.page_size = page_size
        self.remaining_pages = max_pages
        self.space = max_pages * self.page_size
        self.process_list = []


def firstFit(process, linked_list, queue):
    """
    Takes the first block from the linked list which is greater than or equal
    to the requested size. If it can't find one it calls on the second chance
    algorithm to free up some blocks until process can be added.
    """
    added = False
    current_node = linked_list.get_first_node()
    # iterates through list, either reaching the end or stopping if hitting the if statement
    while current_node is not None:
        current_block = current_node.element
```

```
        # if the process fits in the remaining space of a block
        if current_block.space >= process.size:
            # process list is used to find the process later
            current_block.process_list.append(process)
            print("Process with ID %i and size %i KB added to block with max pages of %i." % (process.id, process.size,
current_block.max_pages))
            # math.ceil rounds the number up to see how many pages are needed to fit the process
            process.used_pages = math.ceil(process.size / current_block.page_size)
            print("-> Pages used by this process = %i." % process.used_pages)
            current_block.remaining_pages -= process.used_pages
            print("-> Pages remaining in this block = %i." % current_block.remaining_pages)
            current_block.space = (current_block.remaining_pages * current_block.page_size)
            # above line would not work correctly if there were no remaining pages (multiplying by 0)
            if current_block.remaining_pages == 0:
                current_block.space = 0
            added = True
            # second chance queue
            queue.enqueue(process)
            # stops iterating through list
            current_node = None
        else:
            current_node = current_node.next
    if not added:
        # if something can't be added and the second chance queue is empty then the process is too big
        if queue.length() == 0:
            print("Error: Process with ID %i and size %i KB is too large for any block and was not added." % (process.id,
process.size))
        else:
            secondChance(queue, linked_list)
            # tries to add the process again, will recursively call until added
            firstFit(process, linked_list, queue)


def deAllocation(process, linked_list):
    """
    Process is found in the linked list and removed from the block.
    """
    current_node = linked_list.get_first_node()
    # iterates through list in same way as first fit
    while current_node is not None:
        current_block = current_node.element
        # finding the process in the list
        if process in current_block.process_list:
            print("Process with ID %i and size %i KB de-allocated." % (process.id, process.size))
            current_block.remaining_pages += process.used_pages
            print("-> Pages remaining in this block = %i." % current_block.remaining_pages)
            current_block.space = (current_block.remaining_pages * current_block.page_size)
            # process is still in second chance queue, so this tells the second chance that its done with
            process.access_bit = 0
            current_block.process_list.remove(process)
            current_node = None
        else:
            current_node = current_node.next


def secondChance(queue, linked_list):
    """
    Page replacement managed by a queue. Removes page from queue and checks
    access bit. If its 1, then its set to 0 and page is added back to the queue.
    If its 0, then the page is de-allocated form the linked list.
    """
    process = queue.dequeue()
    if process.access_bit == 0:
        deAllocation(process, linked_list)
    else:
        process.access_bit = 0
        print("Process with ID %i and size %i KB given second chance, added back to queue." % (process.id, process.size))
        queue.enqueue(process)


def simulation(ll_specs, processes_list):
    """
    Simulates all the algorithms working together. Creates a linked list of memory
    blocks using given values, and runs first fit using given processes.
    """
    sll = SLinkedList()
    queue = Queue()
    for tuple in ll_specs:
```

```
        i = 0
        # adds how many of each block wanted for the linked list
        while i < tuple[0]:
            # tuple[1] is how many pages per block, 4 is how many KB per page
            sll.add_first(Block(tuple[1], 4))
            i += 1

    for item in processes_list:
        firstFit(item, sll, queue)
```

Task 4:

Code called as followed:

```
# specifications for creating the linked list; 48 blocks of 2 pages,
# 32 blocks of 4 pages etc. Order is backwards since linked list nodes are
# added to the front of the list.
ll_specs = [(12, 32), (16, 16), (20, 8), (32, 4), (48, 2)]

# creating all the processes
p1 = Process(1, 4)
p2 = Process(2, 4)
p3 = Process(3, 6)
p4 = Process(4, 8)
p5 = Process(5, 12)
p6 = Process(6, 2)
p7 = Process(7, 12)
p8 = Process(8, 16)
p9 = Process(9, 24)

p_list = [p1, p2, p3, p4, p5, p6, p7, p8, p9]

# run sim
simulation(ll_specs, p_list)

print()
print("----- Second chance demonstration -----")
print()

p10 = Process(10, 128)
p11 = Process(11, 128)
p12 = Process(12, 128)
p13 = Process(13, 128)
p14 = Process(14, 128)
p15 = Process(15, 128)
p16 = Process(16, 128)
p17 = Process(17, 128)
p18 = Process(18, 128)
p19 = Process(19, 128)
p20 = Process(20, 128)
p21 = Process(21, 128)
p22 = Process(22, 128)
p23 = Process(23, 128)
p24 = Process(24, 128)

p_list2 = [p10, p11, p12, p13, p14, p15, p16, p17, p18, p19, p20, p21, p22, p23, p24]

simulation(ll_specs, p_list2)
```

Following results printed:

```
caller (2)
/usr/bin/python3.8 "/home/peadar/Labs/CS2506 OS/lab2/caller.py"
Process with ID 1 and size 4 KB added to block with max pages of 2.
-> Pages used by this process = 1.
-> Pages remaining in this block = 1.
Process with ID 2 and size 4 KB added to block with max pages of 2.
-> Pages used by this process = 1.
-> Pages remaining in this block = 0.
Process with ID 3 and size 6 KB added to block with max pages of 2.
-> Pages used by this process = 2.
-> Pages remaining in this block = 0.
Process with ID 4 and size 8 KB added to block with max pages of 2.
-> Pages used by this process = 2.
-> Pages remaining in this block = 0.
Process with ID 5 and size 12 KB added to block with max pages of 4.
-> Pages used by this process = 3.
-> Pages remaining in this block = 1.
Process with ID 6 and size 2 KB added to block with max pages of 2.
-> Pages used by this process = 1.
-> Pages remaining in this block = 1.
Process with ID 7 and size 12 KB added to block with max pages of 4.
-> Pages used by this process = 3.
-> Pages remaining in this block = 1.
Process with ID 8 and size 16 KB added to block with max pages of 4.
-> Pages used by this process = 4.
-> Pages remaining in this block = 0.
Process with ID 9 and size 24 KB added to block with max pages of 8.
-> Pages used by this process = 6.
-> Pages remaining in this block = 2.
```

```
----- Second chance demonstration -----

Process with ID 10 and size 128 KB added to block with max pages of 32.
-> Pages used by this process = 32.
-> Pages remaining in this block = 0.
Process with ID 11 and size 128 KB added to block with max pages of 32.
-> Pages used by this process = 32.
-> Pages remaining in this block = 0.
Process with ID 12 and size 128 KB added to block with max pages of 32.
-> Pages used by this process = 32.
-> Pages remaining in this block = 0.
Process with ID 13 and size 128 KB added to block with max pages of 32.
-> Pages used by this process = 32.
-> Pages remaining in this block = 0.
Process with ID 14 and size 128 KB added to block with max pages of 32.
-> Pages used by this process = 32.
-> Pages remaining in this block = 0.
Process with ID 15 and size 128 KB added to block with max pages of 32.
-> Pages used by this process = 32.
-> Pages remaining in this block = 0.
Process with ID 16 and size 128 KB added to block with max pages of 32.
-> Pages used by this process = 32.
-> Pages remaining in this block = 0.
Process with ID 17 and size 128 KB added to block with max pages of 32.
-> Pages used by this process = 32.
-> Pages remaining in this block = 0.
Process with ID 18 and size 128 KB added to block with max pages of 32.
-> Pages used by this process = 32.
-> Pages remaining in this block = 0.
Process with ID 19 and size 128 KB added to block with max pages of 32.
-> Pages used by this process = 32.
-> Pages remaining in this block = 0.
Process with ID 20 and size 128 KB added to block with max pages of 32.
-> Pages used by this process = 32.
-> Pages remaining in this block = 0.
```

```
Process with ID 21 and size 128 KB added to block with max pages of 32.
-> Pages used by this process = 32.
-> Pages remaining in this block = 0.
Process with ID 10 and size 128 KB given second chance, added back to queue.
Process with ID 11 and size 128 KB given second chance, added back to queue.
Process with ID 12 and size 128 KB given second chance, added back to queue.
Process with ID 13 and size 128 KB given second chance, added back to queue.
Process with ID 14 and size 128 KB given second chance, added back to queue.
Process with ID 15 and size 128 KB given second chance, added back to queue.
Process with ID 16 and size 128 KB given second chance, added back to queue.
Process with ID 17 and size 128 KB given second chance, added back to queue.
Process with ID 18 and size 128 KB given second chance, added back to queue.
Process with ID 19 and size 128 KB given second chance, added back to queue.
Process with ID 20 and size 128 KB given second chance, added back to queue.
Process with ID 21 and size 128 KB given second chance, added back to queue.
Process with ID 10 and size 128 KB de-allocated.
-> Pages remaining in this block = 32.
Process with ID 22 and size 128 KB added to block with max pages of 32.
-> Pages used by this process = 32.
-> Pages remaining in this block = 0.
Process with ID 11 and size 128 KB de-allocated.
-> Pages remaining in this block = 32.
Process with ID 23 and size 128 KB added to block with max pages of 32.
-> Pages used by this process = 32.
-> Pages remaining in this block = 0.
Process with ID 12 and size 128 KB de-allocated.
-> Pages remaining in this block = 32.
Process with ID 24 and size 128 KB added to block with max pages of 32.
-> Pages used by this process = 32.
-> Pages remaining in this block = 0.

Process finished with exit code 0
```