CS2506 Operating Systems II

Main Memory Management

LAB3

Colin Kelleher – 117303363

1. Analysis

- Page Replacement Algorithm = First in First out

First in First Out(FIFO) : This is created using a queue, when we add a page, if there is already pages present, the new pages are added to the end (the tail). Pages are used for a finite period of time

- Memory is made up of several blocks containing different amounts of pages, making each block a different size.

- Size of Main Memory: 1024

- Memory Allocation: Next Fit
List treated as if it were circular and wraps around
gives a more even allocation of memory, allocates free block which is next on the list

- All process requests contain an ID and the number of pages

- During the process, some pages/ blocks can be removed and are free for use again

- Should the system not be able to deal with requests, a message will be printed to the screen, and dealt appropriately within the code

- We start by supplying the blocks of different sizes to the system

2. Design

MainMemory.py – PSEUDOCODE

```
'''
MainMemory.py

Colin Kelleher - 117303363

CS2506 Operating Systems 2 – Lab3 – Main Memory Management
'''
*LLN = Linked List Node
Class Available Memory
     def __init__ (self)
          head = LLN
          tail – LLN (None, head, None)
          last pointer
          first pointer
          size = 0
```

```
    getHead
        return Head
    setHead
        update Head with new head

    getTail
        return Tail
    setNextNode
        update Tail with new Tail

    getFirst
        return First
    setFirst
        update First with new First



    getLast
        return last
    setNextNode
        update Last with new Last

    getSize
        return Size
    setSize
        update size with new Size

    def add (item)
         if the item being added is the first item to list
             call add_first
        otherwise create new node
        the new node is equal to the next node after the last node in the list
        the previous item before the new node is the last node
        the last node is now equal to the new node
        print message to say item was added

    def add_first (item)
        create the node - DLL (item, head, tail)
        first = item (as only item in list)
        last = item (as only item in list)
        increase size by one
        print message to say item is added

    def removeNode (node)
        A = get the node before the node being removed
        B = get the node after the node being removed
```

```
            A = B (previous = next)
            B.previous = A.next
            Decrement the size by one
            Clear the node
            Print message to say it was removed

Class doubly linked list
      def __init__(self, next, previous, memory block)
      pointer to next node
      pointer to previous node
      memory block object

      getNextNode
            return NextNode
      setNextNode
            update nextNode with new next node

      getPreviousNode
            return Previous node
      setPreviousNode
            update previousnode with new previous node

      getMemoryBlock
            return MemoryBlock
      setMemoryBlock
            update Memory block with new memory block
Class Block:
      def __init__ (number of pages)
      number of pages initialisation
      free or not (Boolean), True if block is free for
allocation

Class Main Memory
      def__init__ (memory, pageReplacement)
      memory - initialise free memory
      queue = Page Replacement
      Processes in a key, value dictionary of process_id :
process request

      def requestProcessing (process id, request)
      the process id references specific process in processes
which is equal to the result


      def cleanBlocks (block, pages, process id)
            for each block in the list of blocks
                  remove the specific blocks from memory
            merge pages from deleted blocks to create new block
                  update free to be false
            add block to free memory
            create a dictionary of processes and iterate
through it using keys and values
        run process now that the memory is updated and the
request is satisfied

    def runProcess (process id, block)
        add block to page replacement queue
        return and print message

    def checkMemory
        for each process id in processes
            print message
        get the first in memory
        get the process id
        print message
        if the request is bigger than 1024
            create a dictionary of keys and values, and
for each item in the dictionary if the key is not the process
id, print message
        while there is a next block
            if the next block is free
                if the block pages are <= request
                call RunProcess
                update free to be false
                iterate through dictionary as before
        if there is no next block - no available memory
            check the page replacement queue

Class PageReplacement
    def __init__()
        create list
        head - index of head
        tail - index of tail
        size - size of queue

    def addPageReplacement (block)
        if the queue is empty
            append it to the list
            increment size
            set tail to one
        otherwise
            append to list
            increment size

    def dequeue
        if size = 0
            print message saying queue is empty
            block = head of queue
```

```
          set block to none
          if removing last element in the queue
               reset size, head, tail
          otherwise, if the head is at the end of the list
               we have wrapped around so set head index to be
zero and decrement the size
          otherwise
               increment head and decrement size

     def findPages (id, request)
          if the queue is empty
               print message
          get head of queue
          get pages
          get a list of blocks to be cleaned up
          print message
          create counter
          while counter < length of queue and pages < request
               block = counter index of body
               append block to required
               append pages
               increment counter
          for value in the length of required
               dequeue
          print message
          if the length of requires is greater than one
               return required and pages

     def Kernel
          while there is processes present
               run them

def Test()

     ***testing code goes here***
```

3. Programming

MainMemory.py

```
'''
MainMemory.py

Colin Kelleher - 117303363

CS2506 Operating Systems 2 – Lab3 – Main Memory Management
'''

class AvailableMemBlock:#Defines the class of available
memory, represented as a doubly linked list
    def __init__(self): #initialising
        self._head = LinkedListNode(None, None, None)
#creating head node
        self._tail = LinkedListNode(None, self._head, None)
#creating tail node
        self._head._next = self._tail #the item after the head
is currently the tail
        self._last = None  #last pointer
        self._first = None #first pointer
        self._size = 0 #size

    def getHead(self): #getter for head
        return self._head #return value in head
    def setHead(self, newHead): #setter for head,
        self._head = LinkedListNode(newHead, newHead, newHead)
#update head with new value

    def getTail(self):#getter for tail
        return self._tail #return value in tail
    def setTail(self, newTail): #setter for tail
        self._tail = LinkedListNode(newTail, self._head,
newTail)#update tail with new value

    def getFirst(self): #getter for First
        return self._first #return value in First
    def setFirst(self, newFirst): #setter for First
        self._first = newFirst #update first with new value

    def getLast(self): #getter for Last
        return self._last #return value in Last
    def setLast(self, newLast): #setter for Last
        self._last = newLast #update last with new value

    def getSize(self): #getter for Size
```

```
        return self._size #return value in Size
    def setSize(self, newSize): #setter for Size
        self._size = newSize #update last with new value

    def add(self, item): #defining add method
        if self._first == None: #if the First is empty (equal
to none)
            self.add_first(item) #call add_first method
        else: #otherwise if it is not empty
            newNode = LinkedListNode(item, None, self._tail)
#create node
            self._last._next = newNode #item after last node
is new node
            newNode._prev = self._last #node before new node
is equal to last node
            self._last = newNode #last pointer now equal to
new node, as it was last added
        self._size += 1 #increase size by one
        print("Item added to end of linked list")


    def add_first(self, item): #adds the first item to the
list
        node = LinkedListNode(item, self._head, self._tail)
#creates the node
        self._first = node #this node is the first item list
        self._last = node #this node is also the last item in
the list
        self._size += 1 #increase size by one
        print("Item added to list") #print message


      def removeNode(self, node): #removes node from the list
        previousNode = node._prev #get the previous node
(before node to be removed)
        nextNode = node._next #get the next node (after node
to be removed)
        previousNode._next = nextNode #the previous node is
now equal to the next node
        nextNode._prev = previousNode #node before the
nextnode is now the previous node
        #jumping over removed node
        self._size -= 1 #decrement size by one
        node = LinkedListNode(None, None, None) #clear node
        print('Item removed from linked list') #print message
to say node is removed

class Block: #representing a BLOCK (number of pages)
    def __init__(self, num_of_pages): #initialising
```

```
        self._pages = num_of_pages # number of pages equals
the size of the block
        self._free = True #True if the block is free for
allocation, false otherwise

class LinkedListNode: #defining the structure DLL node
    def __init__(self, block, prev, nextnode):
        self._next = nextnode # pointer to next node
        self._prev = prev # pointer to previous node
        self._block = block # block object

    def getNextNode(self):#getter for Node
        return self._next #return value in next
    def setNextNode(self, nextN): #setter for Next Node
        self._next = nextN #update next with new value

    def getPrevNode(self): #getter for Previous Node
        return self._prev #return value in prev
    def setPrevNode(self, previousN): #setter for Previous
Node
        self._prev = previousN #update previous with new value

    def getBlock(self): #getter for Block
        return self._block #return value in block
    def setBlock(self, newBlock): #setter for Block
        self._block = newBlock #update block with new value


class MainMemory: #creating MAIN MEMORY
    def __init__(self, free_memory, pageReplacement):
#initialising
        self._memory = free_memory #initialising memory
        self._queue = pageReplacement #initialising page
replacement
        self._processes = {} #Here I created a dictionary
containing process_id : process_request key,value pairs


    def requestProcessing(self, process_id, request):
#initialsing a request
        self._processes[process_id] = request #get process
from processes using ID which is equal to the request

    def cleanBlocks(self, blocks, pages, process_id): #tidy up
remove/ merge deleted blocks and pages
        for block in blocks:
            self._memory.removeNode(block) # remove blocks
from free memory
```

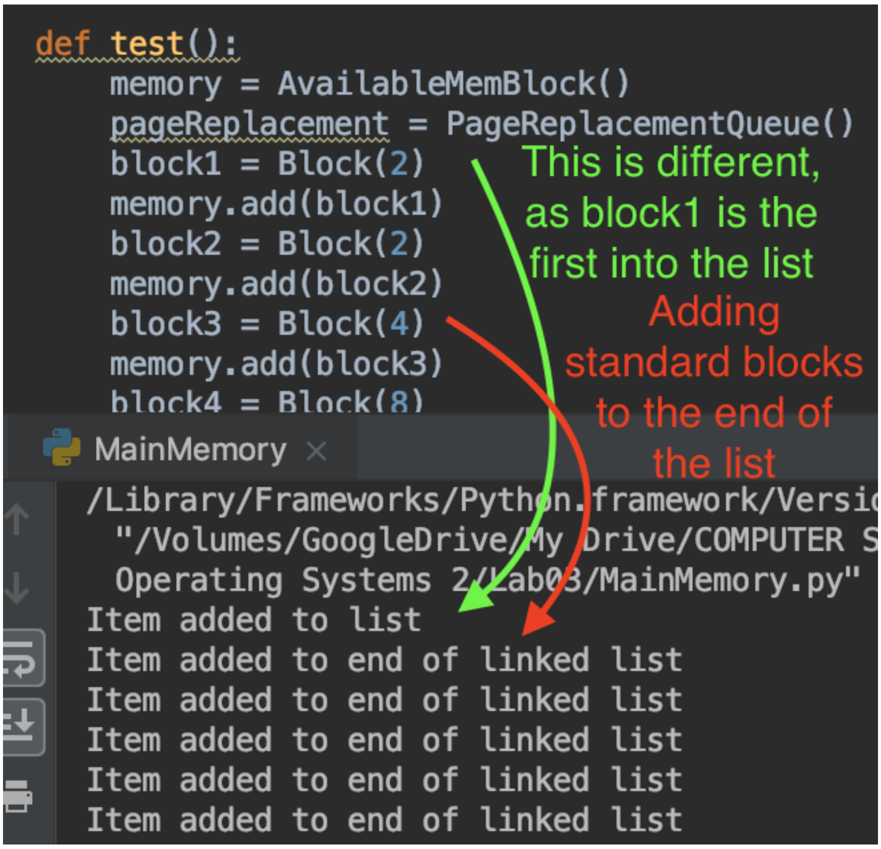
```
        newBlock = Block(pages) # merge pages from deleted
blocks to create new block
        newBlock._free = False #update free to be false
        self._memory.add(newBlock) # add to free memory
        self._processes = {k :v for k ,v in
self._processes.items() if k is not process_id} #create a
dictionary and iterate through the processes using keys and
values
        self.runProcess(process_id, newBlock)#run process now
that memory request satisfied

    def runProcess(self, process_id, block): #Run the Process
        self._queue.addPageReplacement(block) #Add the block
to the page Replacement Queue
        print("Request successful! ; Process %i running in
main memory." % process_id)
        return #print and return the message above

    def checkMemory(self): #check Memory - FIFO algorithm
        for process_id in self._processes: #for each process
in processes
            print("Looking for memory for process %i" %
process_id) #print message
            block = self._memory._first #get the first
            request = self._processes[process_id] #request is
equal to the process id
            print("requested pages: %i" % request) #print
message
            if request > 1024: #if the request is bigger than
1024
                self._processes = {k :v for k ,v in
self._processes.items() if k is not process_id} #create a
dictionary of keys and values, then iterate through it and if
the key is not equal to the process id, print the message as
below
                print("Process request greater than size of
main memory.")
                return
            # find available memory for process request
            while block._next: #while there is a next block
                if block._block._free: #if the block is free
                    if block._block._pages >= request: #if the
block pages are <= request
                        self.runProcess(process_id, block)
#call RunProcess
                        block._block._free = False #update
free to be false
```

```
                        self._processes = {k :v for k ,v in
self._processes.items() if k is not process_id} #iterate
through dictionary as previous
                        break #break out of the code
                block = block._next
                if block._next == None: #if there is no next
                    (blocks, pages) =
self._queue.findPages(process_id, request) #if no available
memory check page replacement queue
                    self.cleanBlocks(blocks, pages,
process_id)#merge blocks found from page replacement queue

class PageReplacementQueue: #create a FIFO page replacement
queue using aspects of Queue AFT
    def __init__(self):
        self.body = [] #create a list
        self.head = 0  #index of head (first element in queue)
        self.tail = 0  #index of tail
        self.size = 0  #size of the queue


    def addPageReplacement(self, block): #add pages/blocks to
the queue
        if self.size == 0:
            self.body.append(block)  # assumes an empty queue
has head at 0
            self.size = 1
            self.tail = 1
        else:
            self.body.append(block)
            self.size += 1

    def dequeue(self): #remove pages from the replacement
queue
        if self.size == 0: #if the queue is empty, print
message
            print("Queue is empty ; No pages to replace
with.")
            return
        block = self.body[self.head] #head of queue
        self.body[self.head] = None
        if self.size == 1:  #if removing last element in list
            self.head = 0 #reset head
            self.tail = 0 #reset tail
            self.size = 0 #reset size
        elif self.head == len(self.body) - 1:#if the head is
now at the end of the list
            self.head = 0 #we have wrapped around in a circie,
set index of head to be zero
```

```
            self.size = self.size - 1 #decrement the size
        else:
            self.head = self.head + 1  #increment head
            self.size = self.size - 1 #decrement size
        return block

    # Finds pages for process request
    def findPages(self, process_id, request): #get pages for
processing
        if self.size == 0: #if the queue is emptyu
            print("Queue empty ; no pages") #print message
            return
        block = self.body[self.head] #get head of list
        pages = block._block._pages #get pages
        required = [block] # list containing blocks to be
cleaned up
        i = 0 #counter
        print("Searching page replacement queue.") #print
message
        while i < len(self.body) and pages < request: #less
than length of queue and pages less than requests
            block = self.body[i] #block = index i of list
            required += [block] #append to required
            pages += block._block._pages #append to pages
            i += 1 #increment counter
        for i in range(len(required)): #for value in range of
length of required
            self.dequeue() # dequeue required blocks from page
replacement queue
        print("Pages replacement successful.")
        if len(required) > 1: #if the length is greater than 1
            return required, pages
        return

def Kernel(main): #Kernel
    while len(main._processes) is not 0: #while there is
processes present
        main.checkMemory() #call checkMemory


if __name__ == "__main__":
    test() #only run the test block, if running the main
program
```

4. Testing

```
'''
MainMemoryManagement.py

Colin Kelleher - 117303363

CS2506 Operating Systems 2 – Lab3 – Main Memory Management
'''
```

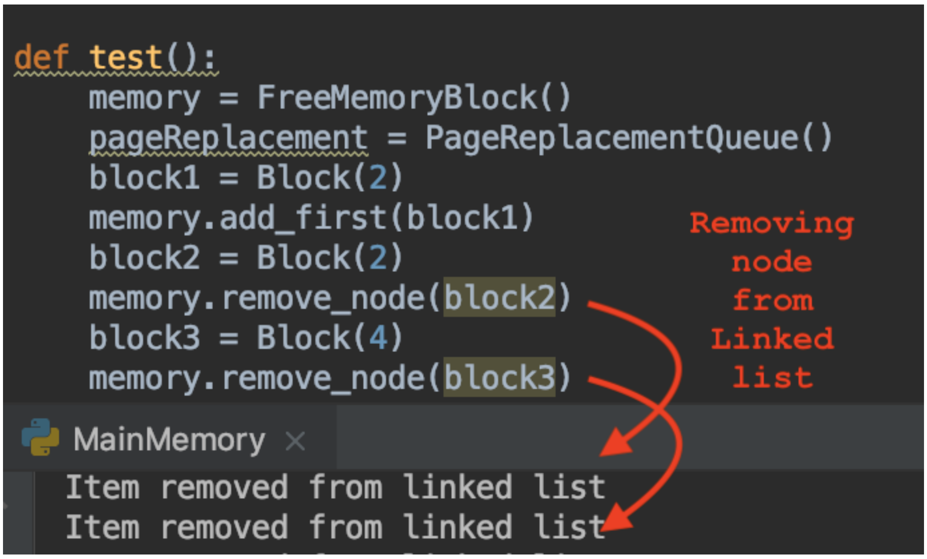


```
def test():
    memory = AvailableMemBlock()
    pageReplacement = PageReplacementQueue()
    block1 = Block(2)
    memory.add(block1)
    block2 = Block(2)
    memory.add(block2)
    block3 = Block(4)
    memory.add(block3)
    block4 = Block(8)
```

This is different, as block1 is the first into the list

Adding standard blocks to the end of the list

```
/Library/Frameworks/Python.framework/Versic
  "/Volumes/GoogleDrive/My Drive/COMPUTER S
  Operating Systems 2/Lab03/MainMemory.py"
Item added to list
Item added to end of linked list
Item added to end of linked list
Item added to end of linked list
Item added to end of linked list
Item added to end of linked list
```

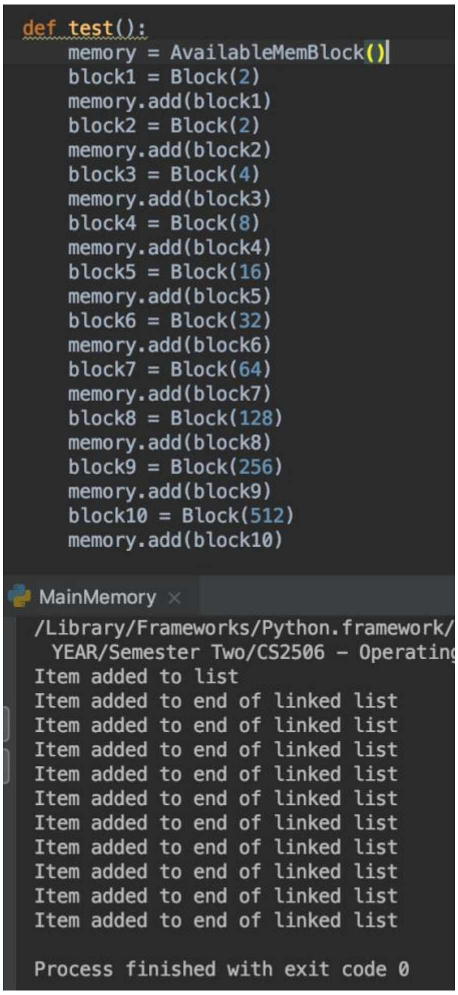


```
def test():
    memory = FreeMemoryBlock()
    pageReplacement = PageReplacementQueue()
    block1 = Block(2)
    memory.add_first(block1)
    block2 = Block(2)
    memory.remove_node(block2)
    block3 = Block(4)
    memory.remove_node(block3)
```

Removing node from Linked list

```
Item removed from linked list
Item removed from linked list
```

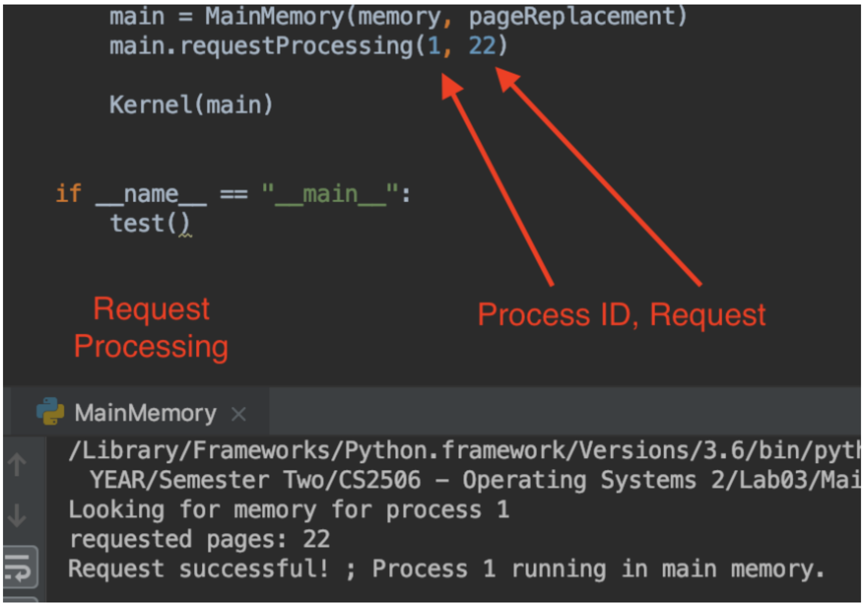
Removing node from linked list



```
def test():
    memory = AvailableMemBlock()
    block1 = Block(2)
    memory.add(block1)
    block2 = Block(2)
    memory.add(block2)
    block3 = Block(4)
    memory.add(block3)
    block4 = Block(8)
    memory.add(block4)
    block5 = Block(16)
    memory.add(block5)
    block6 = Block(32)
    memory.add(block6)
    block7 = Block(64)
    memory.add(block7)
    block8 = Block(128)
    memory.add(block8)
    block9 = Block(256)
    memory.add(block9)
    block10 = Block(512)
    memory.add(block10)
```

```
/Library/Frameworks/Python.framework/
 YEAR/Semester Two/CS2506 - Operating
Item added to list
Item added to end of linked list
Item added to end of linked list
Item added to end of linked list
Item added to end of linked list
Item added to end of linked list
Item added to end of linked list
Item added to end of linked list
Item added to end of linked list
Item added to end of linked list
Item added to end of linked list

Process finished with exit code 0
```

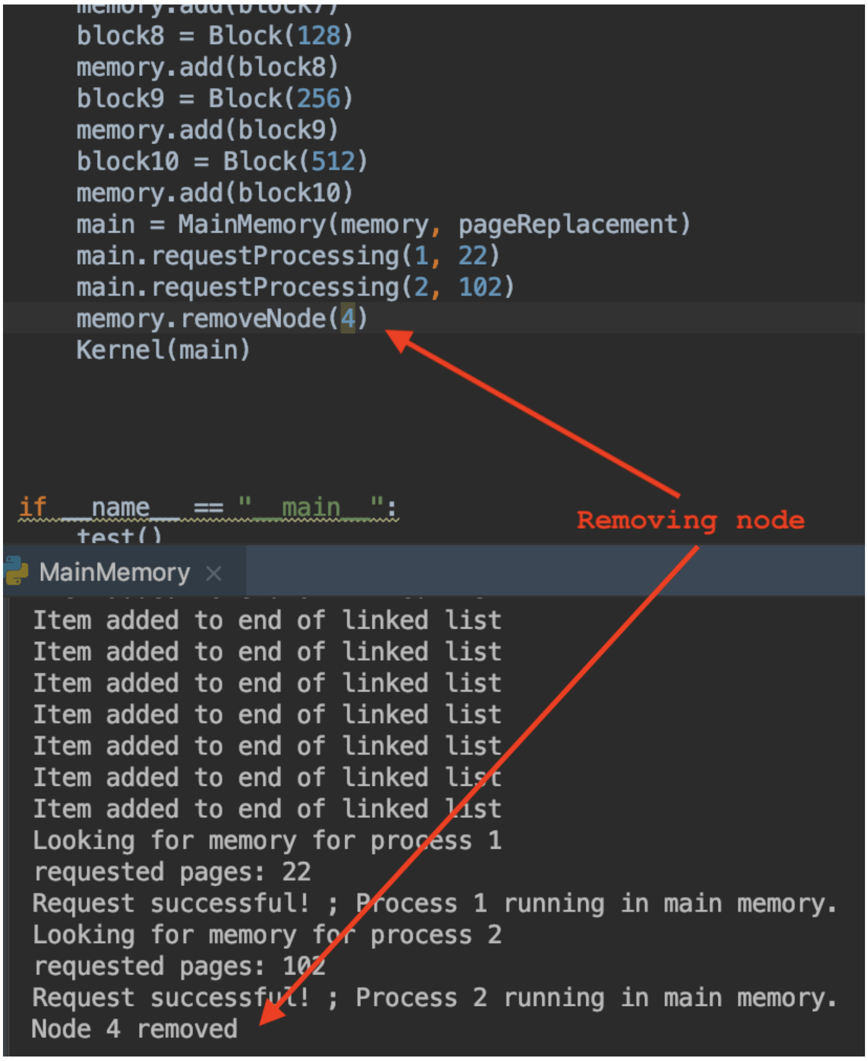
Adding blocks



```
    main = MainMemory(memory, pageReplacement)
    main.requestProcessing(1, 22)

    Kernel(main)


if __name__ == "__main__":
    test()
```

```
/Library/Frameworks/Python.framework/Versions/3.6/bin/pyth
 YEAR/Semester Two/CS2506 - Operating Systems 2/Lab03/Mai
Looking for memory for process 1
requested pages: 22
Request successful! ; Process 1 running in main memory.
```

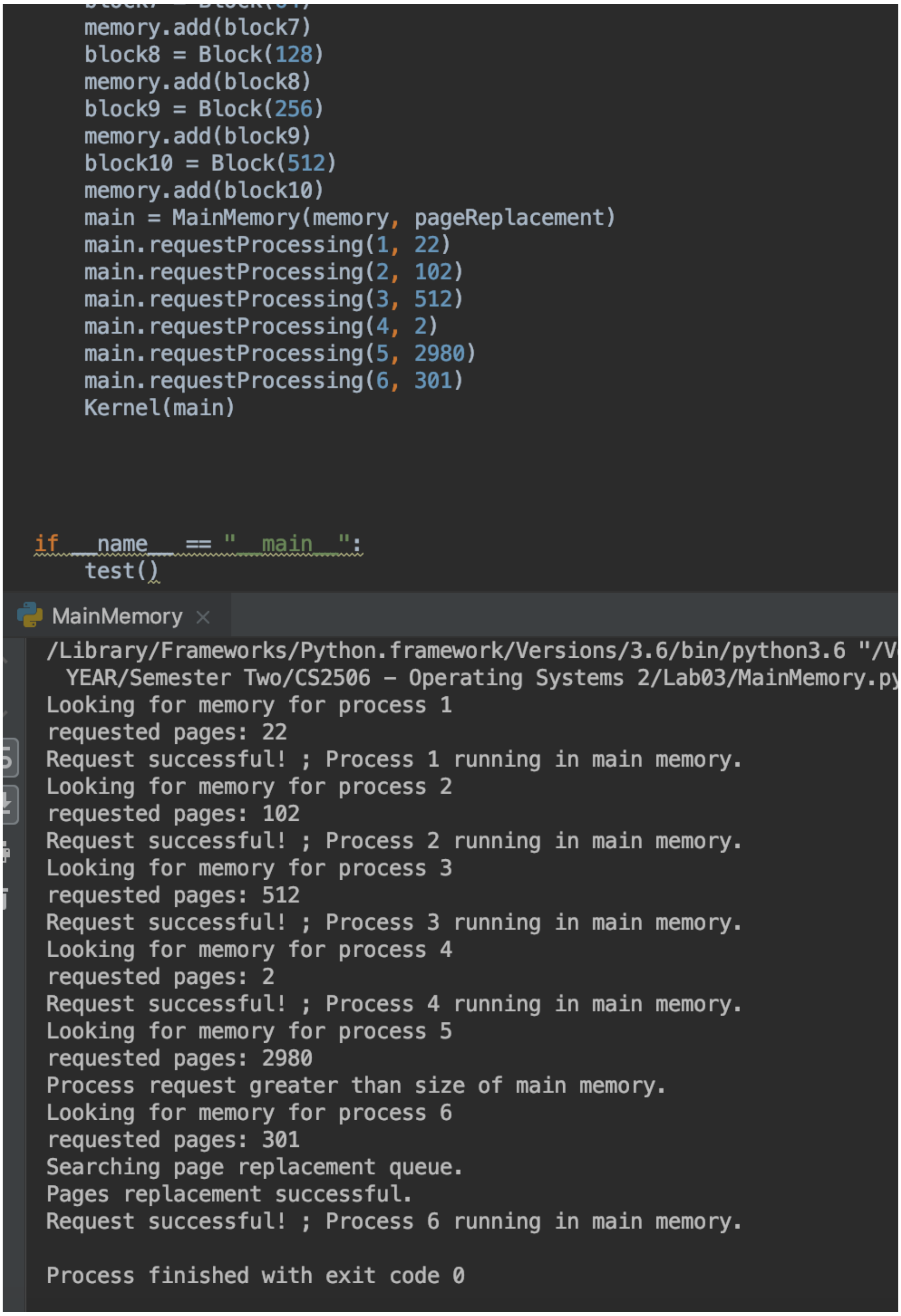
Request Processing



```
    memory.add(block7)
    block8 = Block(128)
    memory.add(block8)
    block9 = Block(256)
    memory.add(block9)
    block10 = Block(512)
    memory.add(block10)
    main = MainMemory(memory, pageReplacement)
    main.requestProcessing(1, 22)
    main.requestProcessing(2, 102)
    memory.removeNode(4)
    Kernel(main)


if __name__ == "__main__":
    test()
```

```
Item added to end of linked list
Item added to end of linked list
Item added to end of linked list
Item added to end of linked list
Item added to end of linked list
Item added to end of linked list
Item added to end of linked list
Looking for memory for process 1
requested pages: 22
Request successful! ; Process 1 running in main memory.
Looking for memory for process 2
requested pages: 102
Request successful! ; Process 2 running in main memory.
Node 4 removed
```

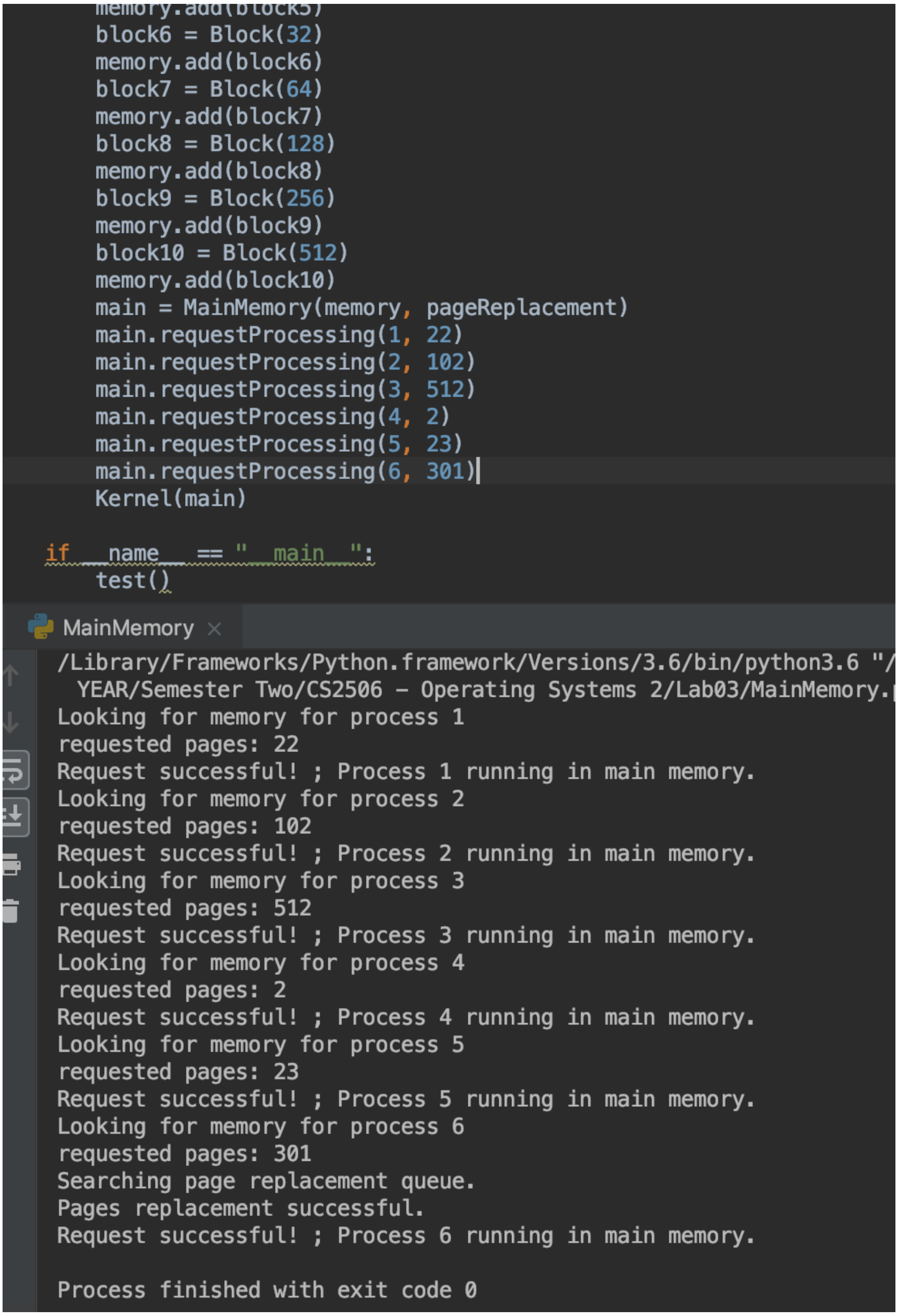
Removing node



```
    block7 = Block(64)
    memory.add(block7)
    block8 = Block(128)
    memory.add(block8)
    block9 = Block(256)
    memory.add(block9)
    block10 = Block(512)
    memory.add(block10)
    main = MainMemory(memory, pageReplacement)
    main.requestProcessing(1, 22)
    main.requestProcessing(2, 102)
    main.requestProcessing(3, 512)
    main.requestProcessing(4, 2)
    main.requestProcessing(5, 2980)
    main.requestProcessing(6, 301)
    Kernel(main)



if __name__ == "__main__":
    test()
```

```
/Library/Frameworks/Python.framework/Versions/3.6/bin/python3.6 "/V
  YEAR/Semester Two/CS2506 - Operating Systems 2/Lab03/MainMemory.py
Looking for memory for process 1
requested pages: 22
Request successful! ; Process 1 running in main memory.
Looking for memory for process 2
requested pages: 102
Request successful! ; Process 2 running in main memory.
Looking for memory for process 3
requested pages: 512
Request successful! ; Process 3 running in main memory.
Looking for memory for process 4
requested pages: 2
Request successful! ; Process 4 running in main memory.
Looking for memory for process 5
requested pages: 2980
Process request greater than size of main memory.
Looking for memory for process 6
requested pages: 301
Searching page replacement queue.
Pages replacement successful.
Request successful! ; Process 6 running in main memory.

Process finished with exit code 0
```

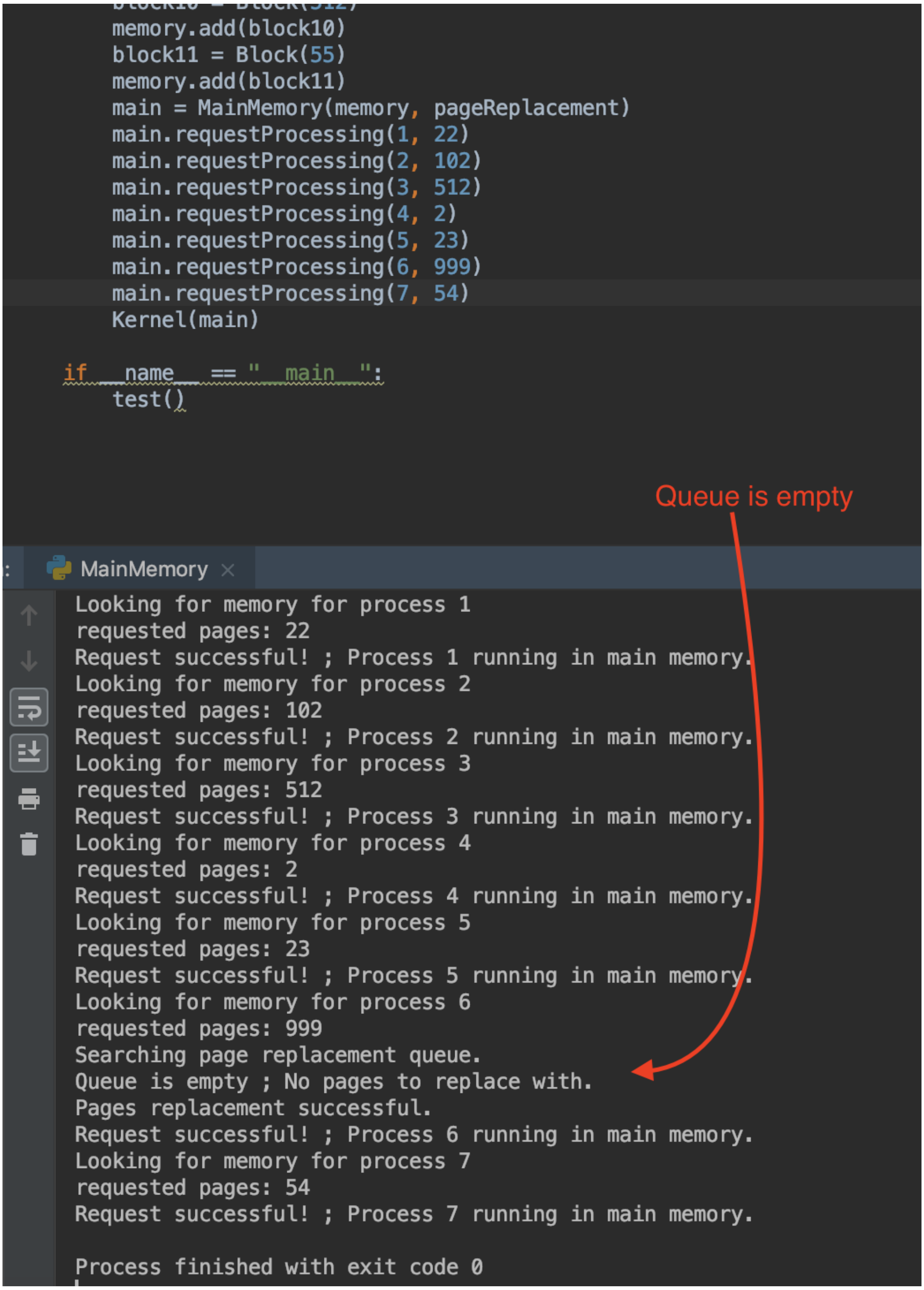
Simulation showing:
- Looking for a process
-Different numbers of pages being requested
- Processes running in main memory
-Replacement queue being used
Process request exceeding the size of main memory



```
    memory.add(block5)
    block6 = Block(32)
    memory.add(block6)
    block7 = Block(64)
    memory.add(block7)
    block8 = Block(128)
    memory.add(block8)
    block9 = Block(256)
    memory.add(block9)
    block10 = Block(512)
    memory.add(block10)
    main = MainMemory(memory, pageReplacement)
    main.requestProcessing(1, 22)
    main.requestProcessing(2, 102)
    main.requestProcessing(3, 512)
    main.requestProcessing(4, 2)
    main.requestProcessing(5, 23)
    main.requestProcessing(6, 301)
    Kernel(main)

if __name__ == "__main__":
    test()
```

```
/Library/Frameworks/Python.framework/Versions/3.6/bin/python3.6 "/
 YEAR/Semester Two/CS2506 - Operating Systems 2/Lab03/MainMemory.
Looking for memory for process 1
requested pages: 22
Request successful! ; Process 1 running in main memory.
Looking for memory for process 2
requested pages: 102
Request successful! ; Process 2 running in main memory.
Looking for memory for process 3
requested pages: 512
Request successful! ; Process 3 running in main memory.
Looking for memory for process 4
requested pages: 2
Request successful! ; Process 4 running in main memory.
Looking for memory for process 5
requested pages: 23
Request successful! ; Process 5 running in main memory.
Looking for memory for process 6
requested pages: 301
Searching page replacement queue.
Pages replacement successful.
Request successful! ; Process 6 running in main memory.

Process finished with exit code 0
```

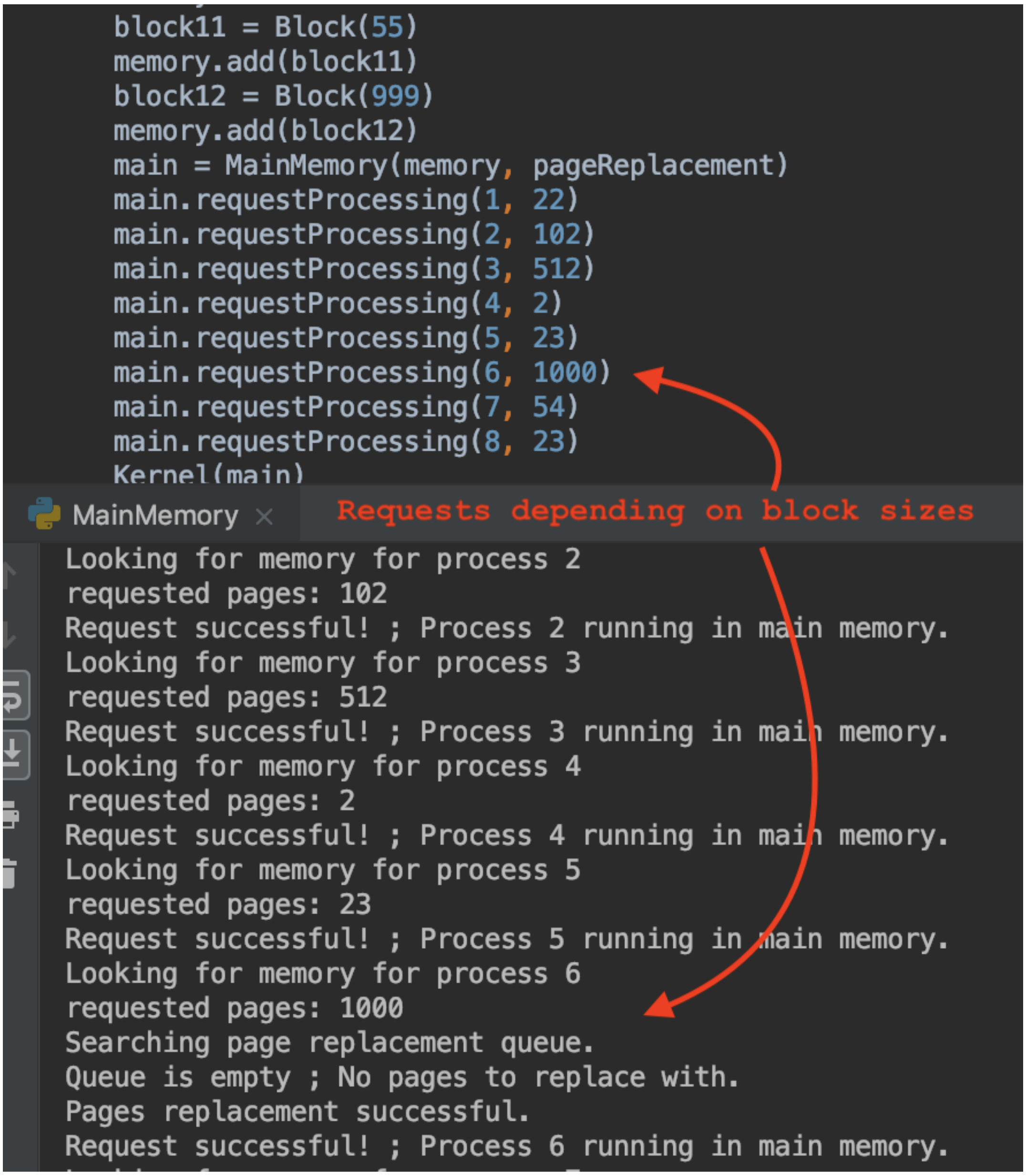
Simulation as above, but memory is not exceeded within this simulation



```
    block10 = Block(512)
    memory.add(block10)
    block11 = Block(55)
    memory.add(block11)
    main = MainMemory(memory, pageReplacement)
    main.requestProcessing(1, 22)
    main.requestProcessing(2, 102)
    main.requestProcessing(3, 512)
    main.requestProcessing(4, 2)
    main.requestProcessing(5, 23)
    main.requestProcessing(6, 999)
    main.requestProcessing(7, 54)
    Kernel(main)

if __name__ == "__main__":
    test()
```

Queue is empty

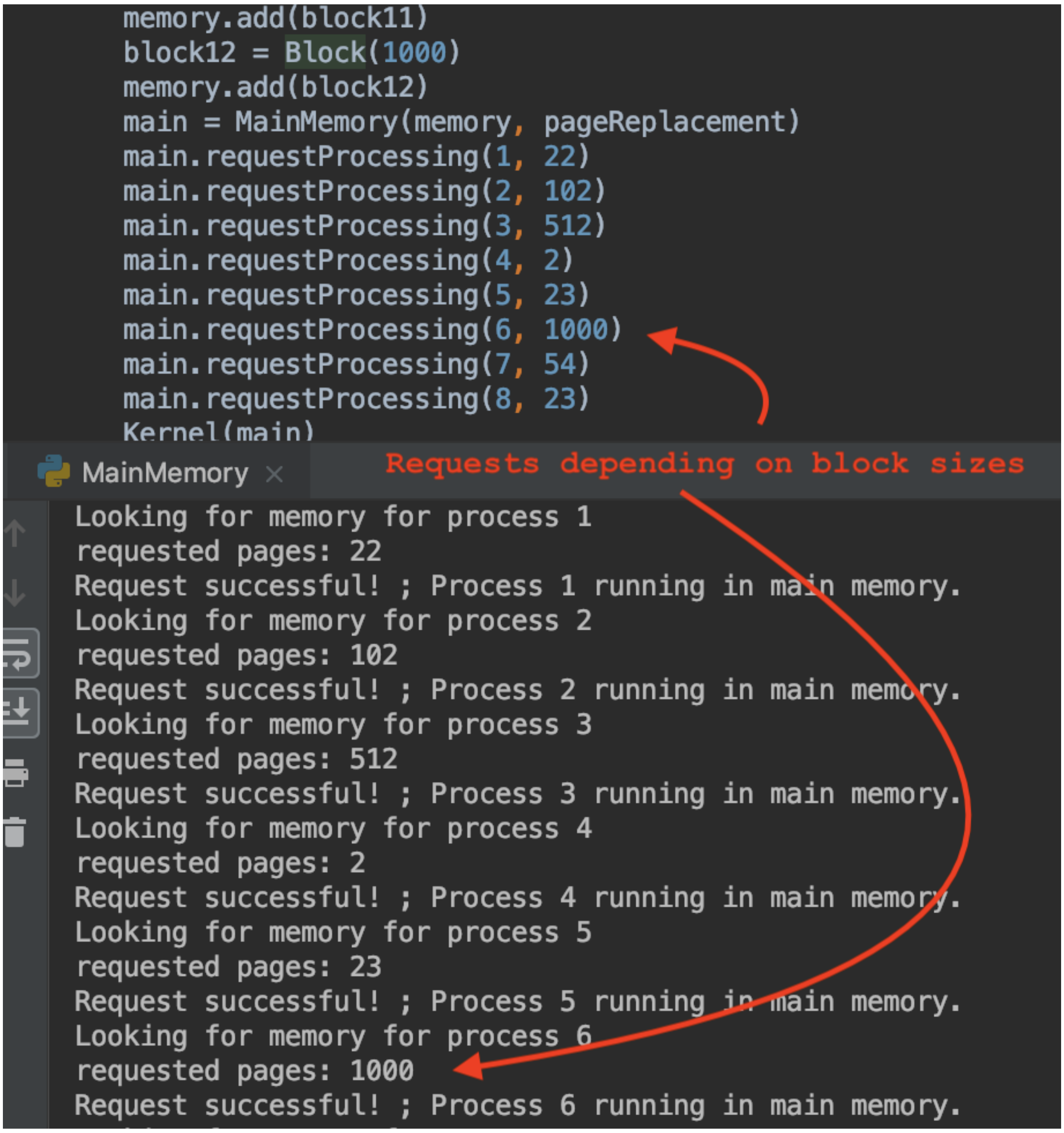
```
Looking for memory for process 1
requested pages: 22
Request successful! ; Process 1 running in main memory.
Looking for memory for process 2
requested pages: 102
Request successful! ; Process 2 running in main memory.
Looking for memory for process 3
requested pages: 512
Request successful! ; Process 3 running in main memory.
Looking for memory for process 4
requested pages: 2
Request successful! ; Process 4 running in main memory.
Looking for memory for process 5
requested pages: 23
Request successful! ; Process 5 running in main memory.
Looking for memory for process 6
requested pages: 999
Searching page replacement queue.
Queue is empty ; No pages to replace with.
Pages replacement successful.
Request successful! ; Process 6 running in main memory.
Looking for memory for process 7
requested pages: 54
Request successful! ; Process 7 running in main memory.

Process finished with exit code 0
```



```
        block11 = Block(55)
        memory.add(block11)
        block12 = Block(999)
        memory.add(block12)
        main = MainMemory(memory, pageReplacement)
        main.requestProcessing(1, 22)
        main.requestProcessing(2, 102)
        main.requestProcessing(3, 512)
        main.requestProcessing(4, 2)
        main.requestProcessing(5, 23)
        main.requestProcessing(6, 1000)
        main.requestProcessing(7, 54)
        main.requestProcessing(8, 23)
        Kernel(main)
```

Requests depending on block sizes

```
Looking for memory for process 2
requested pages: 102
Request successful! ; Process 2 running in main memory.
Looking for memory for process 3
requested pages: 512
Request successful! ; Process 3 running in main memory.
Looking for memory for process 4
requested pages: 2
Request successful! ; Process 4 running in main memory.
Looking for memory for process 5
requested pages: 23
Request successful! ; Process 5 running in main memory.
Looking for memory for process 6
requested pages: 1000
Searching page replacement queue.
Queue is empty ; No pages to replace with.
Pages replacement successful.
Request successful! ; Process 6 running in main memory.
```



```
        memory.add(block11)
        block12 = Block(1000)
        memory.add(block12)
        main = MainMemory(memory, pageReplacement)
        main.requestProcessing(1, 22)
        main.requestProcessing(2, 102)
        main.requestProcessing(3, 512)
        main.requestProcessing(4, 2)
        main.requestProcessing(5, 23)
        main.requestProcessing(6, 1000)
        main.requestProcessing(7, 54)
        main.requestProcessing(8, 23)
        Kernel(main)
```

Requests depending on block sizes

```
Looking for memory for process 1
requested pages: 22
Request successful! ; Process 1 running in main memory.
Looking for memory for process 2
requested pages: 102
Request successful! ; Process 2 running in main memory.
Looking for memory for process 3
requested pages: 512
Request successful! ; Process 3 running in main memory.
Looking for memory for process 4
requested pages: 2
Request successful! ; Process 4 running in main memory.
Looking for memory for process 5
requested pages: 23
Request successful! ; Process 5 running in main memory.
Looking for memory for process 6
requested pages: 1000
Request successful! ; Process 6 running in main memory.
```