Michal Wolas 11730883

Task 1

Main memory: 1MB

size of a page: 4KB

block sizes:

8 blocks of 4 or 6 or 8 or 10 pages, 16 blocks of 2 pages (to allow best fit to better allocate blocks) == all add up to 1024 KB

Page : a page has the allocated bit (in addition to being in the allocated queue) and the accessed bit, the page size

process : id, number of KB , pages own access bit

block : owns pages, has allocated bit and accessed bit

free memory tracking algorithm:

dictionary of all free pages.
key = the block size
value = list of block instances with the same amount of pages.

taken/ Allocated pages are placed into a round robin queue (a list)

memory allocation algorithm:

best fit algorithm
given to algorithm - list of available block sizes (initially always [2,4,6,8,10])

- the process size

returns the list of block sizes best suited for the process size

example - > request for 23KBs

pages = ceiling of 23/4 = 6
matches 6 pages required with block of 6 pages , returns 6 pages

example - > request for 50KBs
pages = ceiling of 50/4 = 13
matches 10 on the first iteration
next iteration it needs to match 3 pages
matches 4
returns [10,4]

page replacement algorithm:

second chance algorithm

triggered when memory is low and process requests more than available.

given - the size to free up, the process pointer

goes through the taken/allocated list, round robin style. Checks the allocated bit, places the block at then end of the queue if accessed, otherwise frees it up (by placing it in the free dictionary accordingly, updating the allocated bit)

returns the result of the best fit algorithm, to be ready to be allocated to the process

how the interact:

when a process makes a request. Available space is checked. if there is sufficient space, best fit algorithm is called. (works as above, returns a list of blocks)

the process is now allocated the blocks returned by best fit. Each block allocated bit is updated and the process now owns those blocks. (it can access them)

if there isn't sufficient memory:
Page replacement is called. (works as above) .

the process owns the blocks and can access them. changing the allocated bit, i chose a 70% chance of any block being accesses. This access bit is checked when Second chance algorithm is called.

data structures:

classes : Page, Process, Block , MainMemory, KernelServices
block has a page/s
process has a block. can change blocks access bit
MainMemory - owns blocks. can updates and manages them
KernelServices control the blocks and main memory

list: list are fast in python, and easy to maintain

dictionary:
used to store free pages.

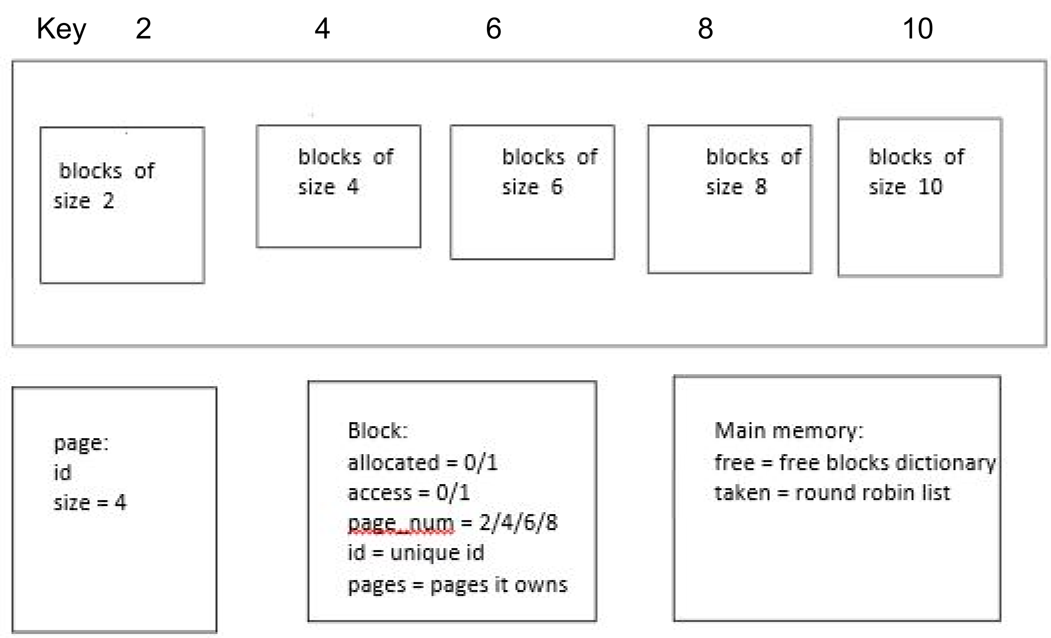
I chose this as Best fit needs to be able to tell the program what blocks to allocate fast . One of the main downsides of best fit is that it needs to perform a lot of searching, dictionary structure was my attempt at reducing that complexity.
with Keys being O(1) look up, the lists my algorithm is working with are very small due to having these keys .
instead of searching through all the blocks, i search through available sizes, and can easily tell the program to pop or add to appropriate keys values all in O(1)

why these data structures:

I chose dictionaries to quickly find the sizes of block i want in O(1). It also orders the blocks by their size at the keys. This attempts to combat the best fit inefficiency at searching the list of free blocks

I chose lists inside the dictionaries are pythons appending and removing is efficient, as well as searching.

Diagram of data structures



Pseudocode

Class Page: attributes - size, id

Class Block : attributes - id , numberOfPages, allocatedbit, accessbit, pages

Def KB (): return numberOfPages * 4

Getters for size and pages

Def populate: creates instances of pages, based on number of pages

Class MainMemory: attributes - free, taken

Def create():
Create 8 blocks of 4, 6,8 and 10
Create 10 blocks of 2

Def addtoTaken(block): adds a block to the taken list and updates allocated bit

Def addtoFree(block): adds a lock to free and updates the allocated bit
Def freeKBs(): report how many free KBs there are

    For each list in the dictionary:
        For each block in that list:
            Free += block.KB()

    Return free

Def filledUp():

    Reports true or false if the main memory is full

Def freeBlocks(): returns list of free block sizes

    For each list:
        If that list has blocks
            Add to list

    Return list

Class Process: attributes- id, size, blocks(list of blocks owned)

Def allocateblock(block) : adds the block to the list of owned blocks

Def accesstheBlock(block):

70% chance of accessing the block

Change blocks access bit to 1

Class KernelServices: attributes - mainMemory

Def bestFit(process):

Size =proces.size

Freeblocks = list of key sizes of free blocks

```
    numPages = cieling of size/4




    Blocks = []
    While numPages >0:
        If the pages required can be satisfied perfectly from available blocks:
            Append amount to blocks
            Return blocks
        Elif if the pages requested is greater than biggest block size(10):
            Add 10 to blocks, if available
            Decrement from numpages


        else:
            Find the smallest biggest block that satisfies the size
            Add it to the blocks
            Decrement from numpages

            If you cant find smallest biggest:
                This means memory is low, allocate left over blocks

    Return blocks


Def allocate(process)
    Check if the there is enough space for the block
    If there isn't space : run second chance algorithm
    If there is run bestFit to find out the blocks

    For each of the block sizes:
        Remove an instance of that size from memory
        Give it to the process
        Add it to taken list ( using addtoTaken(block) )
```

Def secondChance(tofreeup, process):

Freed = 0

While freed < tofreeup:

If block at index 0 in taken was not accessed:
Add block to free (using addtoFree(block))
Freed += block.KB()
Remove that block from taken

Else:
Give it second chance:
Take it out of taken
Change accessbit to 0
Append it to taken

Python Code:

```
import random
import math
import time


# represents a page, owned by a block. Has id and size
class Page:
   def __init__(self, id):
       self.size = 4
       self.id = id

   def __str__(self):
       return str(self.id)


# Block class owns Pages of variable sizes, The main memory Owns the blocks
and stores them in a data structure
```

```python
class Block:
    def __init__(self, id, numberOfPages):
        self.id = id
        self.numberOfPages = numberOfPages
        self.allocatedbit = None
        self.accessbit = None
        self.pages = []

    # returns the number of pages this block owns
    def size(self):
        return self.numberOfPages

    # returns the KBs this block can store
    def KB(self):
        return self.numberOfPages * 4

    # Boolean checker, whether the block was accessed or not ( used by Second
chance algorithm )
    def accessed(self):
        if self.accessbit == 1:
            return True
        return False

    # Returns a string showing the pages it has, to show the main memory
structure ( part of mainMemory create method )
    def getPages(self):
        pagesList = ""
        for i in self.pages:
            pagesList += str(i) + ", "

        return pagesList[:-2]

    def __str__(self):
        return str(self.id)

    # populates the block with instances of pages
    def populate(self):
        for page in range(self.numberOfPages):
            self.pages += [Page(str(self.id) + "." + str(page))]
```

```python
class MainMemory:
    def __init__(self):
        # dictionary where keys are the number of pages a block has
        # value is a list of blocks all of which have the same number of pages
        self._free = {2: [], 4: [], 6: [], 8: [], 10: []}

        # round-robin style list, items are appended at the end, and taken out
the front
        self._taken = []

    # creates all the necessary instances of blocks building up self._free
structure
    def create(self):
        # creates 8 blocks of different predefined sizes
        for block in range(8):
            new_block4 = Block("four." + str(block), 4)
            new_block4.populate()
            self._free[4].append(new_block4)

            new_block6 = Block("six." + str(block), 6)
            new_block6.populate()
            self._free[6].append(new_block6)

            new_block8 = Block("eight." + str(block), 8)
            new_block8.populate()
            self._free[8].append(new_block8)

            new_block10 = Block("ten." + str(block), 10)
            new_block10.populate()
            self._free[10].append(new_block10)

        # creates 16 blocks of 2 pages, this is for increased best fit
algorithm precision
        for block in range(16):
            new_block2 = Block("two." + str(block), 2)
            new_block2.populate()
            self._free[2].append(new_block2)
```

```python
        self.represent()

    # prints all the blocks ( and pages inside the blocks ) that the main
memory has
    def represent(self):
        for x in self._free:
            for y in self._free[x]:
                print("block: ", y, " has pages: ", y.getPages())

        print(self)
        print()

    # call issued by kernel allocate method
    # Updates the blocks allocated bit and appends it to the taken list
    def addtoTaken(self, block):
        block.allocatedbit = 1
        self._taken.append(block)

    # issued by the kernel secondChance method
    # Updates the blocks allocated bit and appends it to the free dictionary
accordingly
    def addtoFree(self, block):
        block.allocatedbit = 0
        self._free[block.size()].append(block)

    # reports the of free KBs left in the main memory
    def freeKBs(self):
        total = 0
        for k in self._free:
            for block in self._free[k]:
                total += block.KB()
        return total

    # report True if the main memory is almost full, false otherwise
    def filledUp(self):
        if self.freeKBs() <= 30:  # considering 30kbs as already filled up , as
i chose process size to be 2KBs - 40KBss
            return True
```

```python
        return False

    # check how many blocks of a particular size are left
    def freeBlocksofSize(self, size):
        return len(self._free[size])

    # returns a list of free KEYS that can be allocated
    # by keys i mean the block of a certain size
    # if freeblock() returns  [2,6,8] -> this mean that only blocks of size 2,
6 and 8 are available

    def freeBlocks(self):
        free = []
        for k in self._free:
            if len(self._free[k]) > 0:
                free += [k]
        return free

    def __str__(self):
        free = self.freeKBs()
        description = "Main Memory size:  1024KB \tavailable: " + str(free) +
"KB"
        return description


class Process:
    # process has, id, the size it will require in KBs
    def __init__(self, id, size):
        self.id = id
        self.size = size
        self.blocks = []  # the blocks this process was allocated

    def __str__(self):
        return str(self.id)

    # called by kernel allocate method
    # Process requests a block and is allocated a block
    def allocateBlock(self, block):
        self.blocks += [block]
```

```python
        # each time it has 70% chance to write to the  block making it accessed
        self.accessTheBlock(block)

    # 70% chance ( chosen arbitrarily ) of a block being accessed by the
process
    def accessTheBlock(self, block):
        x = random.randint(1, 10)

        if x <= 7:
            block.accessbit = 1  # update the blocks access bit


class KernelServices:

    def __init__(self):
        # create and populate the main memory
        self.mainMemory = MainMemory()
        self.mainMemory.create()

    # Best fit algorithm , It is asked to allocate blocks to a process
    # it returns a list of blocks sizes e.g
    # [4] , this tells "def allocate" that the process is best fitted for 1
block of 4
    # if memory is scare it might return something like [2,2,2] , meaning 3
blocks of 2.
    def bestFit(self, process):
        # given a process it returns a list of blocks sizes best suited for the
process size
        size = process.size

        # turns size into minimum number of pages required
        numPages = math.ceil(size / 4)
        freeBlocks = self.mainMemory.freeBlocks()
        blocks = []
        while numPages > 0:
            # if the requested size is available in perfectly
            if numPages in freeBlocks:
                blocks += [numPages]
                return blocks
```

```
            # if the size is greater than or equal to the max block size
            elif numPages >= 10 and 10 in freeBlocks:
                blocks += [10]
                numPages -= 10
            else:
                try:
                    # tries to assign the correct block
                    # examples
                    #y = 4 , if request for 3 pages is made
                    # if request for 7 pages was made, y = 8
                    y = min(filter(lambda x: x > numPages, freeBlocks))
                    numPages -= y
                    blocks += [y]
                except: # this exception occurs when y failed to match a block
                    # this could happen if 7 pages was requested and 10 and 8
sized blocks are taken
                    # in that case any block is added ,as this happens only
towards the end, about the last 50KBs or so
                    if self.mainMemory.freeKBs() >= process.size:
                        # to make sure that we are not allocating too many
blocks of one size than there are available
                        # for example, to prevent trying to allocate 3 blocks
of 2, if we only have 2 blocks of 2
                        if not blocks.count(freeBlocks[0]) >
(self.mainMemory.freeBlocksofSize(freeBlocks[0])) + 1:
                            numPages -= freeBlocks[0]
                            blocks += [freeBlocks[0]]
                    else: # safety measure to ensure correct flow of execution
                        return self.secondChance(process.size, process)
        return blocks

    # allocates memory to a process, uses Best fit and Second Chance
    def allocate(self, process):
        # allocate block/s to the process
        print("------- NEW PROCESS REQUESTS BLOCKS --------")
        print("Process ID: ", str(process), "Current: ",
self.mainMemory.freeKBs(), "KBs Needed : ", process.size,
             "KBs\n")
```

```python
        bestFit = []
        if self.mainMemory.freeKBs() < process.size:
            bestFit = self.secondChance(process.size, process)
        else:
        # the array of blocks that the process needs to be allocated based on
best fit
            bestFit = self.bestFit(process)

        for i in bestFit:
            blockToBeAllocated = self.mainMemory._free[i].pop() # removes the
block from free
            process.allocateBlock(blockToBeAllocated) # allocates the block to
a process
            self.mainMemory.addtoTaken(blockToBeAllocated) # adds process to
taken list

    # Page replacement algorithm , Second Chance
    # triggered when a process requests memory and the main memory is full
    def secondChance(self, toFree, process):

        print("\n--------- Page replacement ( second chance ) -------------")
        print("Current free Kbs: ", self.mainMemory.freeKBs(), " Need : ",
toFree, "Kbs")

        x = 0
        freed = 0

        # frees up enough space to allow the process to be allocated memory
        while freed < toFree:
            # checks if an allocated memory, (round robin style), has been
accessed
            if not self.mainMemory._taken[0].accessed():
                # frees up that page
                self.mainMemory.addtoFree(self.mainMemory._taken[0])
                freed += self.mainMemory._taken[0].KB()
                self.mainMemory._taken.pop(0)
                x += 1
```

```
            else: # if it has been accessed it is given a second chance
                giveSecondChance = self.mainMemory._taken.pop(0)
                giveSecondChance.accessbit = 0
                self.mainMemory._taken.append(giveSecondChance)

        print(freed, " KBs have been freed, by de-allocating ", x, "
block/s\n")

        # allow best fit to give the correct blocks to a process
        return self.bestFit(process)

    # reports if the main memory is full or not
    def full(self):
        return self.mainMemory.filledUp()

    # report information about the main memory
    def info(self):
        return str(self.mainMemory)


#
kernel = KernelServices()

i = 0
processList = []
# fill up the memory untill its full
while not kernel.full():
    i += 1
    process = Process(str(i), random.randint(2, 50))
    processList += [process]
    # print(process.size)
    kernel.allocate(process)

print()
print(len(processList), " processes have been allocated so far , filling up
the memory to about 30KBs remaining")
print()

# add processes to show page replacement algorithm
```

```python
while True:
    i += 1
    process = Process(str(i), random.randint(2, 50))
    processList += [process]
    # print(process.size)
    time.sleep(1)
    kernel.allocate(process)
```

Screenshots

```
C:\Users\michal\UniWork\venv\Scripts\python.exe "C:/Users/michal/Un
block:  two.0  has pages:  two.0.0, two.0.1
block:  two.1  has pages:  two.1.0, two.1.1
block:  two.2  has pages:  two.2.0, two.2.1
block:  two.3  has pages:  two.3.0, two.3.1
block:  two.4  has pages:  two.4.0, two.4.1
block:  two.5  has pages:  two.5.0, two.5.1
block:  two.6  has pages:  two.6.0, two.6.1
block:  two.7  has pages:  two.7.0, two.7.1
block:  two.8  has pages:  two.8.0, two.8.1
block:  two.9  has pages:  two.9.0, two.9.1
block:  two.10  has pages:  two.10.0, two.10.1
block:  two.11  has pages:  two.11.0, two.11.1
block:  two.12  has pages:  two.12.0, two.12.1
block:  two.13  has pages:  two.13.0, two.13.1
block:  two.14  has pages:  two.14.0, two.14.1
block:  two.15  has pages:  two.15.0, two.15.1
block:  four.0  has pages:  four.0.0, four.0.1, four.0.2, four.0.3
block:  four.1  has pages:  four.1.0, four.1.1, four.1.2, four.1.3
block:  four.2  has pages:  four.2.0, four.2.1, four.2.2, four.2.3
block:  four.3  has pages:  four.3.0, four.3.1, four.3.2, four.3.3
block:  four.4  has pages:  four.4.0, four.4.1, four.4.2, four.4.3
block:  four.5  has pages:  four.5.0, four.5.1, four.5.2, four.5.3
block:  four.6  has pages:  four.6.0, four.6.1, four.6.2, four.6.3
block:  four.7  has pages:  four.7.0, four.7.1, four.7.2, four.7.3
```

```
block:  six.0  has pages:  six.0.0, six.0.1, six.0.2, six.0.3, six.0.4, six.0.5
block:  six.1  has pages:  six.1.0, six.1.1, six.1.2, six.1.3, six.1.4, six.1.5
block:  six.2  has pages:  six.2.0, six.2.1, six.2.2, six.2.3, six.2.4, six.2.5
block:  six.3  has pages:  six.3.0, six.3.1, six.3.2, six.3.3, six.3.4, six.3.5
block:  six.4  has pages:  six.4.0, six.4.1, six.4.2, six.4.3, six.4.4, six.4.5
block:  six.5  has pages:  six.5.0, six.5.1, six.5.2, six.5.3, six.5.4, six.5.5
block:  six.6  has pages:  six.6.0, six.6.1, six.6.2, six.6.3, six.6.4, six.6.5
block:  six.7  has pages:  six.7.0, six.7.1, six.7.2, six.7.3, six.7.4, six.7.5
block:  eight.0  has pages:  eight.0.0, eight.0.1, eight.0.2, eight.0.3, eight.0.4, eight.0.5, eight.0.6, eight.0.7
block:  eight.1  has pages:  eight.1.0, eight.1.1, eight.1.2, eight.1.3, eight.1.4, eight.1.5, eight.1.6, eight.1.7
block:  eight.2  has pages:  eight.2.0, eight.2.1, eight.2.2, eight.2.3, eight.2.4, eight.2.5, eight.2.6, eight.2.7
block:  eight.3  has pages:  eight.3.0, eight.3.1, eight.3.2, eight.3.3, eight.3.4, eight.3.5, eight.3.6, eight.3.7
block:  eight.4  has pages:  eight.4.0, eight.4.1, eight.4.2, eight.4.3, eight.4.4, eight.4.5, eight.4.6, eight.4.7
block:  eight.5  has pages:  eight.5.0, eight.5.1, eight.5.2, eight.5.3, eight.5.4, eight.5.5, eight.5.6, eight.5.7
block:  eight.6  has pages:  eight.6.0, eight.6.1, eight.6.2, eight.6.3, eight.6.4, eight.6.5, eight.6.6, eight.6.7
block:  eight.7  has pages:  eight.7.0, eight.7.1, eight.7.2, eight.7.3, eight.7.4, eight.7.5, eight.7.6, eight.7.7
block:  ten.0  has pages:  ten.0.0, ten.0.1, ten.0.2, ten.0.3, ten.0.4, ten.0.5, ten.0.6, ten.0.7, ten.0.8, ten.0.9
block:  ten.1  has pages:  ten.1.0, ten.1.1, ten.1.2, ten.1.3, ten.1.4, ten.1.5, ten.1.6, ten.1.7, ten.1.8, ten.1.9
block:  ten.2  has pages:  ten.2.0, ten.2.1, ten.2.2, ten.2.3, ten.2.4, ten.2.5, ten.2.6, ten.2.7, ten.2.8, ten.2.9
block:  ten.3  has pages:  ten.3.0, ten.3.1, ten.3.2, ten.3.3, ten.3.4, ten.3.5, ten.3.6, ten.3.7, ten.3.8, ten.3.9
block:  ten.4  has pages:  ten.4.0, ten.4.1, ten.4.2, ten.4.3, ten.4.4, ten.4.5, ten.4.6, ten.4.7, ten.4.8, ten.4.9
block:  ten.5  has pages:  ten.5.0, ten.5.1, ten.5.2, ten.5.3, ten.5.4, ten.5.5, ten.5.6, ten.5.7, ten.5.8, ten.5.9
block:  ten.6  has pages:  ten.6.0, ten.6.1, ten.6.2, ten.6.3, ten.6.4, ten.6.5, ten.6.6, ten.6.7, ten.6.8, ten.6.9
block:  ten.7  has pages:  ten.7.0, ten.7.1, ten.7.2, ten.7.3, ten.7.4, ten.7.5, ten.7.6, ten.7.7, ten.7.8, ten.7.9
Main Memory size:  1024KB  available: 1024KB
```

```
------- NEW PROCESS REQUESTS BLOCKS --------
Process ID:  1 Current:  1024 KBs Needed :  12 KBs

------- NEW PROCESS REQUESTS BLOCKS --------
Process ID:  2 Current:  1008 KBs Needed :  37 KBs

------- NEW PROCESS REQUESTS BLOCKS --------
Process ID:  3 Current:  968 KBs Needed :  29 KBs

------- NEW PROCESS REQUESTS BLOCKS --------
Process ID:  4 Current:  936 KBs Needed :  17 KBs

------- NEW PROCESS REQUESTS BLOCKS --------
Process ID:  5 Current:  912 KBs Needed :  43 KBs

------- NEW PROCESS REQUESTS BLOCKS --------
Process ID:  6 Current:  864 KBs Needed :  9 KBs

------- NEW PROCESS REQUESTS BLOCKS --------
Process ID:  7 Current:  848 KBs Needed :  46 KBs

------- NEW PROCESS REQUESTS BLOCKS --------
Process ID:  8 Current:  800 KBs Needed :  23 KBs
```

Proccess kept queuing up, untill memory became full
It takes about 30-50 process with randomized KBs range of 2-50KBs

```
------- NEW PROCESS REQUESTS BLOCKS --------
Process ID:  22 Current:  328 KBs Needed :  42 KBs

------- NEW PROCESS REQUESTS BLOCKS --------
Process ID:  23 Current:  280 KBs Needed :  12 KBs

------- NEW PROCESS REQUESTS BLOCKS --------
Process ID:  24 Current:  264 KBs Needed :  48 KBs

------- NEW PROCESS REQUESTS BLOCKS --------
Process ID:  25 Current:  216 KBs Needed :  5 KBs

------- NEW PROCESS REQUESTS BLOCKS --------
Process ID:  26 Current:  208 KBs Needed :  9 KBs

------- NEW PROCESS REQUESTS BLOCKS --------
Process ID:  27 Current:  192 KBs Needed :  11 KBs

------- NEW PROCESS REQUESTS BLOCKS --------
Process ID:  28 Current:  176 KBs Needed :  44 KBs

------- NEW PROCESS REQUESTS BLOCKS --------
Process ID:  29 Current:  128 KBs Needed :  30 KBs

------- NEW PROCESS REQUESTS BLOCKS --------
Process ID:  30 Current:  96 KBs Needed :  38 KBs

------- NEW PROCESS REQUESTS BLOCKS --------
Process ID:  31 Current:  48 KBs Needed :  11 KBs
```

31 processes have been allocated so far , filling up the memory to about 30KBs remaining
Now as memory is full, Page Replacement strategy is used

```
------- NEW PROCESS REQUESTS BLOCKS --------
Process ID:  32 Current:  24 KBs Needed :  41 KBs


--------- Page replacement ( second chance ) -------------
Current free Kbs:  24  Need :  41 Kbs
48  KBs have been freed, by de-allocating  2  block/s

------- NEW PROCESS REQUESTS BLOCKS --------
Process ID:  33 Current:  24 KBs Needed :  37 KBs


--------- Page replacement ( second chance ) -------------
Current free Kbs:  24  Need :  37 Kbs
40  KBs have been freed, by de-allocating  2  block/s

------- NEW PROCESS REQUESTS BLOCKS --------
Process ID:  34 Current:  24 KBs Needed :  3 KBs

------- NEW PROCESS REQUESTS BLOCKS --------
Process ID:  35 Current:  0 KBs Needed :  25 KBs


--------- Page replacement ( second chance ) -------------
Current free Kbs:  0  Need :  25 Kbs
48  KBs have been freed, by de-allocating  3  block/s

------- NEW PROCESS REQUESTS BLOCKS --------
Process ID:  36 Current:  16 KBs Needed :  30 KBs
```

```
------- NEW PROCESS REQUESTS BLOCKS --------
Process ID:  36 Current:  16 KBs Needed :  30 KBs


--------- Page replacement ( second chance ) -------------
Current free Kbs:  16  Need :  30 Kbs
56  KBs have been freed, by de-allocating  2  block/s

------- NEW PROCESS REQUESTS BLOCKS --------
Process ID:  37 Current:  32 KBs Needed :  11 KBs

------- NEW PROCESS REQUESTS BLOCKS --------
Process ID:  38 Current:  16 KBs Needed :  39 KBs


--------- Page replacement ( second chance ) -------------
Current free Kbs:  16  Need :  39 Kbs
56  KBs have been freed, by de-allocating  2  block/s

------- NEW PROCESS REQUESTS BLOCKS --------
Process ID:  39 Current:  32 KBs Needed :  6 KBs

------- NEW PROCESS REQUESTS BLOCKS --------
Process ID:  40 Current:  0 KBs Needed :  29 KBs


--------- Page replacement ( second chance ) -------------
Current free Kbs:  0  Need :  29 Kbs
40  KBs have been freed, by de-allocating  1  block/s

------- NEW PROCESS REQUESTS BLOCKS --------
Process ID:  41 Current:  0 KBs Needed :  39 KBs


--------- Page replacement ( second chance ) -------------
Current free Kbs:  0  Need :  39 Kbs
64  KBs have been freed, by de-allocating  3  block/s
```