Diagram

Description automatically generatedDiagram

Description automatically generated

Python implementation of the Buddy System, AND included memory tracking dictionary

"""

author: Konrad Dagiel 120329501

"""

import math

class Process:

    """

    Processes only need a size for simulation purposes, and an id to be distinguished

    """

    def \_\_init\_\_(self,id,size):

        self.\_size=size

        self.\_id=id

class Node:

    """

    Tree structure means nodes point to their children and parent

    Nodes have a size, and can have a process

    Only leaf nodes are considered to be part of the memory, as

    outlined in the algorithm rundown.

    """

    def \_\_init\_\_(self, size, parent=None, process=None):

        self.\_left=None

        self.\_right=None

        self.\_parent=parent

        self.\_size=size

        self.\_process=process

    def split(self):

        """

        create 2 children, each half size. Since the node is no longer

        a leaf, it is not available to take a process in.

        The children are considered as part of the main memory.

        """

        self.\_left=Node(self.\_size//2,self)

        self.\_right=Node(self.\_size//2,self)

        return self.\_left

    def merge(self):

        """

        wipe a node and its buddy, meaning the parent is now a leaf and

        considered to be part of the main memory

        """

        self.\_parent.\_left=None

        self.\_parent.\_right=None

        return self.\_parent

    def \_\_str\_\_(self):

        return str(self.\_size)

    def getBuddy(self):

        """

        get the current nodes "buddy" aka sibling

        """

        if not self.\_parent:

            return None

        if self==self.\_parent.\_left:

            return self.\_parent.\_right

        elif self==self.\_parent.\_right:

            return self.\_parent.\_left

        return None

    def isLeaf(self):

        """

        test if a node is a leaf.

        """

        return self.\_left is None and self.\_right is None

    def listAvailableLeaves(self):

        """

        list all available spots in the memory

        """

        arr=[]

        self.allAvailableLeaf(arr)

        for i in range(len(arr)):

            arr[i]=arr[i].\_size

        return arr

    def allAvailableLeaf(self, arr):

        """

        helper function to list all available memory locations in an array

        """

        if (not self):

            return

        # Check if current self is a leaf node and doesnt have a process

        if (not self.\_left and not self.\_right):

            if not self.\_process:

                arr.append(self)

            return

        # Traverse the left

        # and right subtree

        self.\_right.allAvailableLeaf(arr)

        self.\_left.allAvailableLeaf(arr)

    def minLeaf(self, spaceRequired):

        """

        returns the minimum size leaf that can fit the requested size.

        """

        arr=[]

        self.allAvailableLeaf(arr)

        minNode=Node(memorySize+1) #minNode initially bigger than memory

        for i in arr:

            if i.\_size>=spaceRequired and i.\_size<minNode.\_size:

                minNode=i

        #if found a suitable node, return. if not, return None

        return minNode if minNode.\_size < memorySize+1 else None

    def allocate(self, process):

        """

        round space to a power of 2

        find min size suitable block

        split repeatedly until its size is the required space

        allocate the process to that block

        """

        print("Allocating process", process.\_id)

        #if block size is lower than minimum, set it to minimum

        #else round it to nearest power of 2

        spaceRequired = minBlockSize if process.\_size < minBlockSize else 2\*\*(math.ceil(math.log(process.\_size, 2)))

        print("space required = ", spaceRequired)

        print("internal fragmentation = ", spaceRequired-process.\_size)

        x=self.minLeaf(spaceRequired)

        #if no min found, cannot allocate

        if not x:

            print("not enough space in memory, not allocated")

            print("----------------------------")

            return None

        #split repeatedly

        while x.\_size // 2 >= spaceRequired:

            print("splitting, new size=", x.\_size//2)

            x=x.split()

        print("found block, size=",x.\_size)

        #update allocated dictionary, so we know which process is in which block

        allocatedDict[process]=x

        #finally, allocate process

        x.\_process=process

        print("available blocks: ", self.listAvailableLeaves())

        print("----------------------------")

        return x

    def deallocate(self, process):

        """

        lookup our dict to find where this process is in memory

        deallocate it, and merge the block it was in if its buddy is free

        merge repeatedy while possible

        """

        print("De-allocating process",process.\_id)

        #can only de-allocate processes that have been allocated

        if process not in allocatedDict:

            print("no such process exists")

            print("----------------------------")

            return None

        x=allocatedDict[process]

        #update dict

        allocatedDict.pop(process)

        print("deallocated process:",process.\_id)

        x.\_process=None

        y=x.getBuddy()

        #repeatedly merge empty leaves

        while not y.\_process and y.isLeaf():

            print("buddy also empty, merging")

            temp=x.\_parent

            x.merge()

            x=temp

            y=x.getBuddy()

            if not y:

                print("available blocks: ", self.listAvailableLeaves())

                print("----------------------------")

                return process

        print("available blocks: ", self.listAvailableLeaves())

        print("----------------------------")

        return process

allocatedDict={} #this is used to track where each process is

memorySize=4096 #this is our initial memory size

minBlockSize=512 #this is the smallest allowed size for a block

root=Node(memorySize) #this is how we initialize our memory

p0=Process(0,1000)

p1=Process(1,1000)

p2=Process(2,1000)

p3=Process(3,3000)

p4=Process(4,400)

processList=[p0, p1, p2, p3, p4] #list of processes to be allocated

print("available blocks:", root.listAvailableLeaves())

print("----------------------------")

for process in processList:

    root.allocate(process)

#random order to deallocate

root.deallocate(p3)

root.deallocate(p2)

root.deallocate(p4)

root.deallocate(p1)

root.deallocate(p0)

Output Explained:

Text

Description automatically generated

Process 0 to be allocated.

P0 needs 1000 units, so we round to the nearest power of 2, 1024.

This results in internal fragmentation of 24 units.

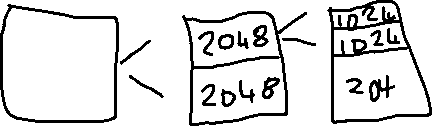
The starting memory size is 4096 units, which means we can split it into two 2048 size blocks.

We then split it again into two 1024 size blocks.

We now found a block that can be used for our request.

This leaves us with 2 available blocks.

The tree evolves like this:



Then we do the same for process 1 and 2:

Text

Description automatically generated

Process 3 is too big to fit into the memory, it cannot be allocated.

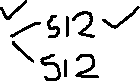
Process 4 is smaller than the minimum size (we set to 512 units for demonstration purpose, but this can be lowered to any power of 2).

This means it is rounded up to 512.

Text

Description automatically generated

We are finished allocating, and the tree looks as follows



We can now start de-allocating. This is done in random order, to simulate different processes needing the memory for different amounts of time.

A picture containing graphical user interface

Description automatically generated

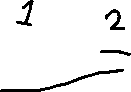
First we try to de allocate process 3, but since process 3 was never allocated, we simply move on.

Text

Description automatically generated

Process 2 is de-allocated, no merging happens, as its buddy is not free

When we de-allocate process 4, two merges happen as follows:



Now we de-allocate the remaining processes, and what we should observe, is the entire memory re-merging into one block.

Text

Description automatically generated

Sure enough, after another two merges, the memory is back to one block of 4069.