**Task 1**

Size of Main Memory (user space): 4MB

Size of a Page: 4KB

Organisation of memory blocks:

* 32 blocks of 8 pages = 1024KB
* 16 blocks of 16 pages = 1024KB
* 8 blocks of 32 pages = 1024KB
* 4 blocks of 64 pages = 1024KB

64 Blocks in total = 4096KB, which encompasses the entire user space

**Algorithms**

**Free Memory Tracking**

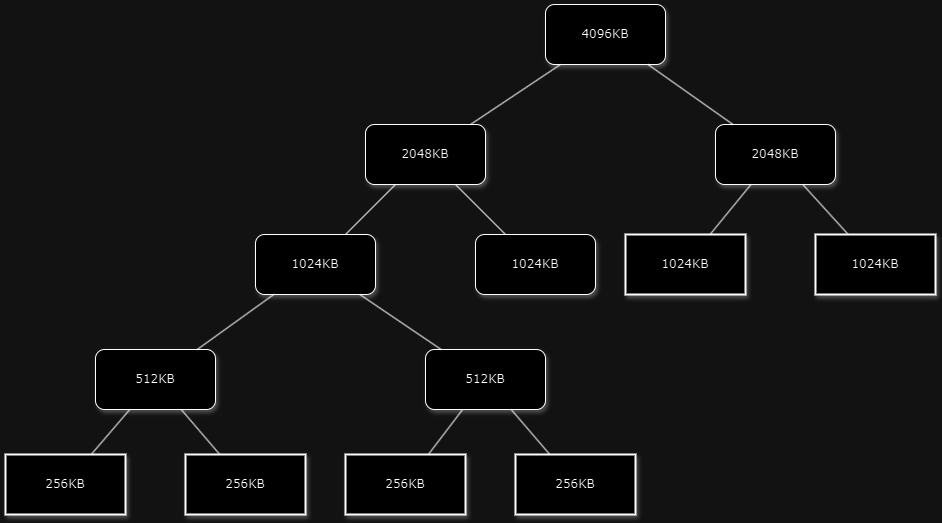
* Due to the fact that the memory is allocated in variable-sized blocks, the data structure that stores the address to the free blocks would need to be highly efficient.
* An algorithm with a fast lookup time (O(1) or O(log n)) would likely be needed to efficiently search the data structure for free blocks of memory.

**Linked Lists**

* Would contain objects that store:
  + First memory address
  + Size
  + Next object
* A linked list wouldn’t be efficient due to its O(n) search time.
* When a memory request is made, the linked list will be searched for a block of closest fit.
* Due to its lookup and search time, I won’t use a linked list, however it should be kept in mind as it could be used in addition to other data structures.

**Binary Trees**

* Binary trees would be far more efficient than Linked lists in terms of lookup and search times, providing O(log n) for worst-case lookup times.
* The lowest possible level of the binary tree would be a level where the block size would be the size of a page, in this case 4KB.
* This data structure would work well with the buddy system in memory allocation, as it makes it very easy to split the blocks.



* In this diagram, there are 4 blocks of 256KB and 3 blocks of 1024KB, which adds up to 4096KB.
* When a request is made for 512KB, it will be given a block of 1024KB.

**Hash Maps**

* Hash Maps would be much more efficient than Linked Lists, due to their fast lookup and search times.
* The keys would be the block sizes, and the values would be the blocks of that size
* Hash Maps could be even more efficient should you make the values Linked Lists containing the blocks.
* For this project, I’ll be using Hash-Maps, not only for their fast look-up and search times, but also for their ease of implementation.
* Though Binary Search Trees might be faster in some cases, they’re much harder to manage and more difficult to code.

**Memory Allocation**

* For memory allocation I’ll be using **Best Fit**.
  + Best Fit searches the entire list and
* The reason I’ll be using Best Fit is due to it working well with Hash Maps, as I can simply select the size of the block I want and instantly get a block of that size.

**Task 2**

Pseudocode

Page Class contains:

* Start address
* Process ID
* Access Bit (for page replacement)
* A method to allocate a process
* Deallocate a process

Block Class

* Variables:
  + List of pages it contains
  + Free Memory, used memory, total memory
  + A dictionary of process IDs and respective pages
* “largest\_run\_of\_free\_pages” method
  + Loop through all pages
  + Find the longest stretch of free pages
* “allocate\_memory(PID: int, size: int)” method
  + Find the largest contiguous free space
  + Allocate the PID to the pages
  + Update memory stats and process allocations
  + Return the allocated pages
    - Or None, if there wasn’t enough space
* “deallocate\_memory(pid: int)” method
  + Deallocate memory assigned to the process
    - By looping through the list of pages
  + Update memory stats and process allocations
  + Return the amount of memory deallocated
* “has\_amount(size: int)”
  + Returns a Boolean to indicate whether or not the block has that much free contiguous space.

MemoryRequest class

* \_\_init\_\_(pid: int, size: int)
  + Sets the pid and size in kb

MemoryManager class

* \_\_init\_\_(memory\_config)
  + Initialise memory\_config (how blocks are organised)
  + Initialise total\_memory, all\_block (list), free\_blocks (dict)
    - The **free\_block** dictionary will store all the free blocks, keys will be block sizes, and the values will be a linked list of blocks of that size
  + Initialise a process dictionary to keep track of what blocks a process is using
* get\_closest\_power\_of\_two(size: int) -> int
  + find the closest power of two smaller than or equal to the given size
* allocate\_memory(request: MemoryRequest):
  + allocate memory to a process by choosing the smallest block with enough free space
  + Then update each memory stats and the process allocation dictionary
  + If a block is not found, call the clock\_algorithm method
* deallocate\_memory(pid: int):
  + By using the process allocation dictionary, we can find all the blocks being used by a process.
  + Loop these blocks are deallocate them and update the memory stats
* clock\_algorithm(request: MemoryRequest):
  + Search through blocks for one with enough total space to hold the request
  + Loop through pages in the block and check the access bit
  + If it hasn’t been accessed in a certain period of time, allow it to be replaced

class OperatingSystem:

* \_\_init\_\_():
  + Initialise memory request queue
    - Using a queue implementation from another module
  + Initialise a MemoryManager object
* add\_memory\_request(request: MemoryRequest):
  + Add the request to the memory request queue
* process\_memory\_requests():
  + keep dequeuing from the queue and allocate the memory through the memory manager
* finish\_process\_execution(pid: int):
  + Using the “deallocate\_memory” method from the memory manager class, we can deallocate memory based off the pid

This pseudocode shows all 3 algorithms working together. With all of them being in the “MemoryManager” class.

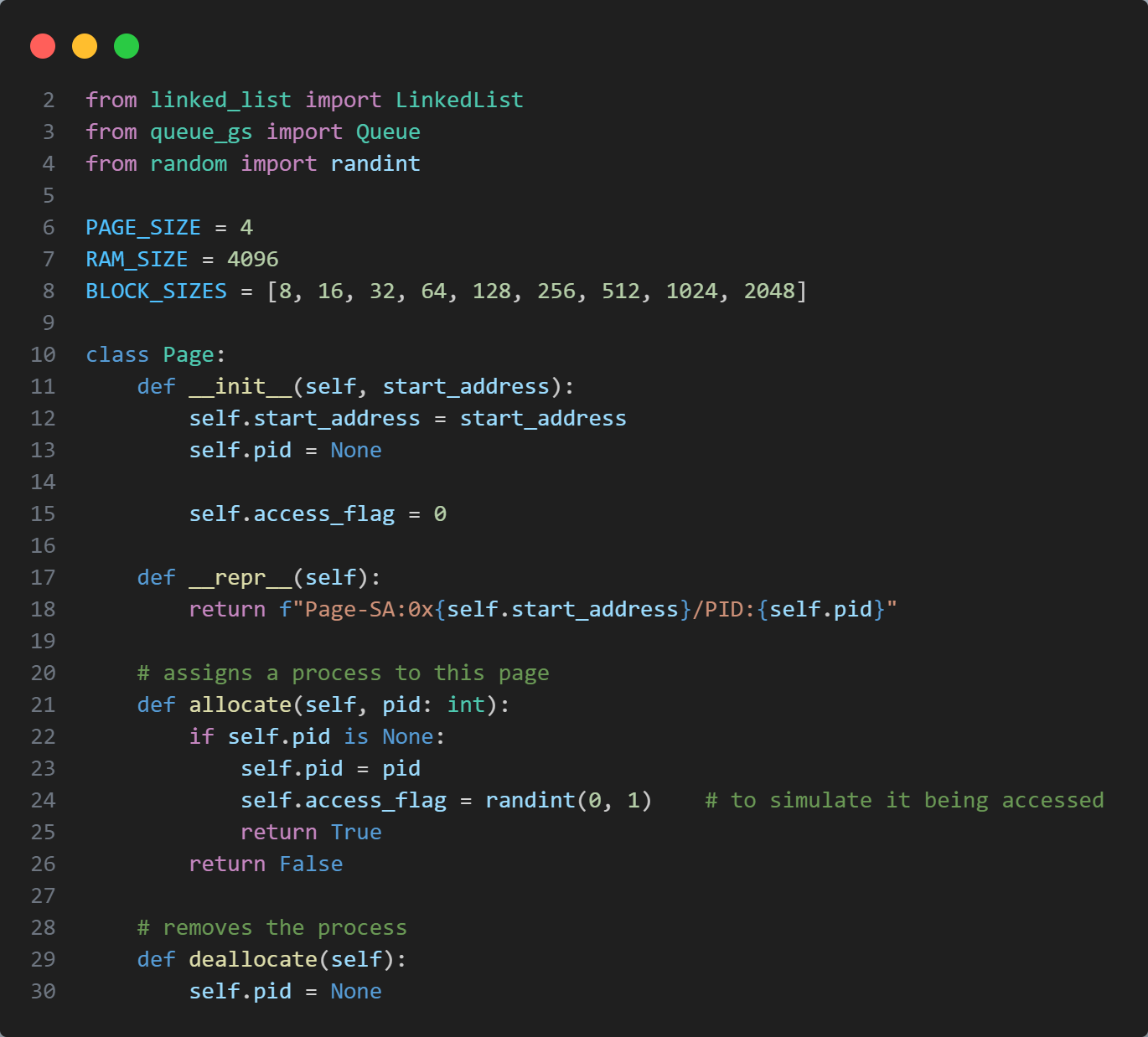
The free memory tracking is done with a dictionary, with memory sizes as the keys and linked lists containing blocks of those sizes as the values.

The **best fit** algorithm is used in the “allocate\_memory” function, in which it takes a size and tries to the find the closest size with the power of 2. It then searches the dictionary and finds a block of that size. If there is no block of that size it will go to the next largest size.

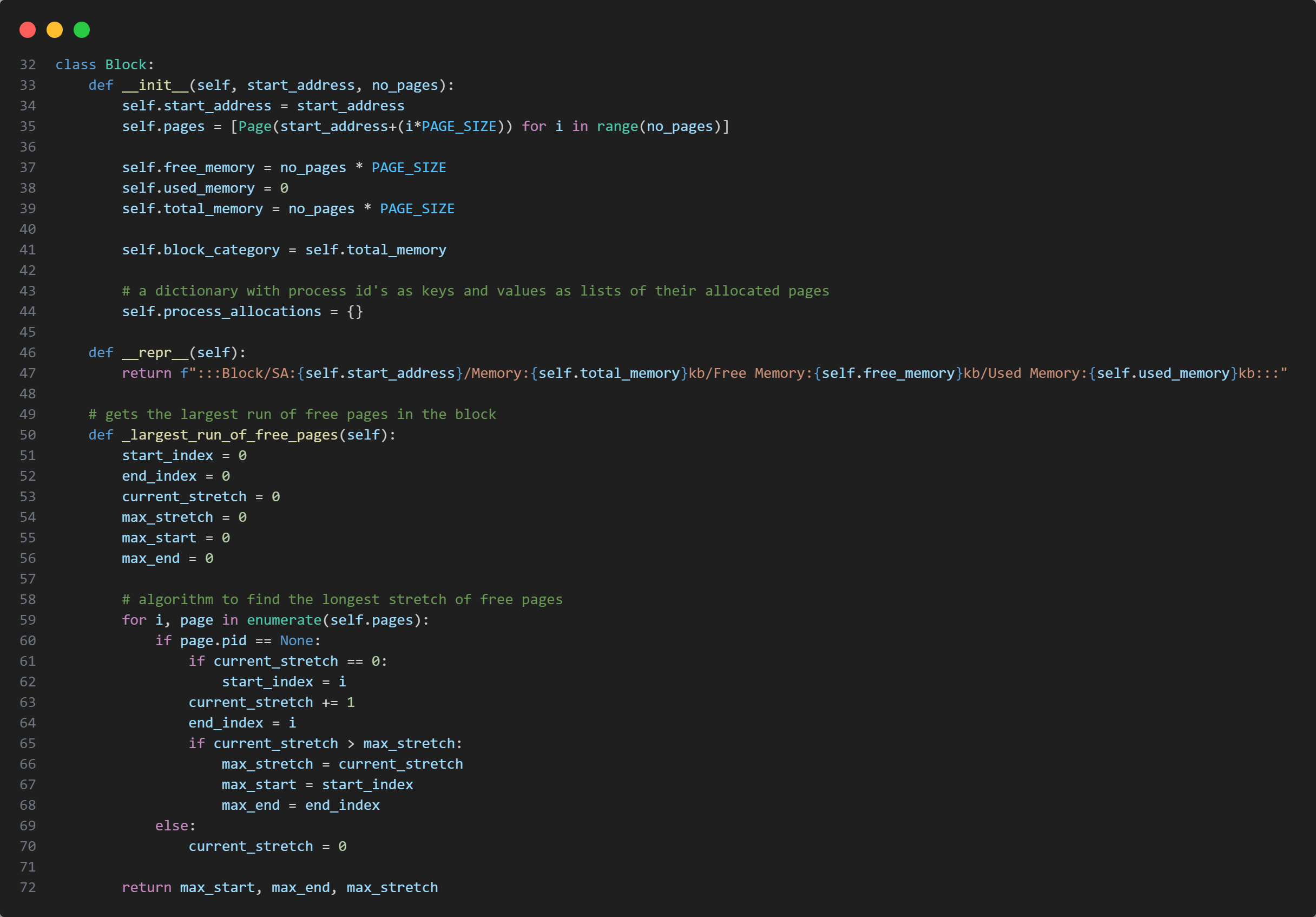
The clock algorithm is used in the “clock\_algorithm” function, which takes in a MemoryRequest and searches through all the blocks, finding one large enough to satisfy the memory request, and it implements the clock algorithm on that block.

**Task 3**

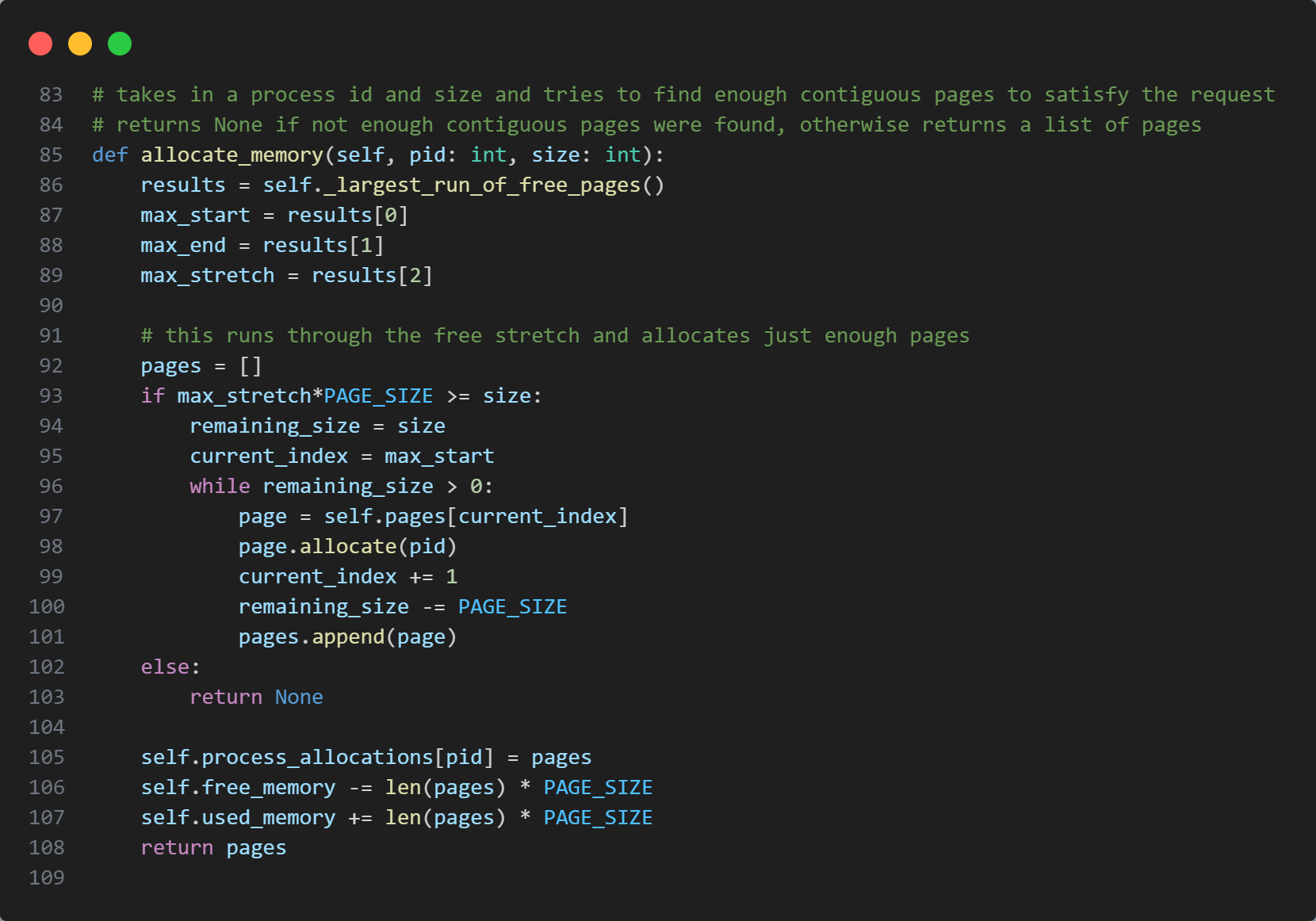
The Page Class

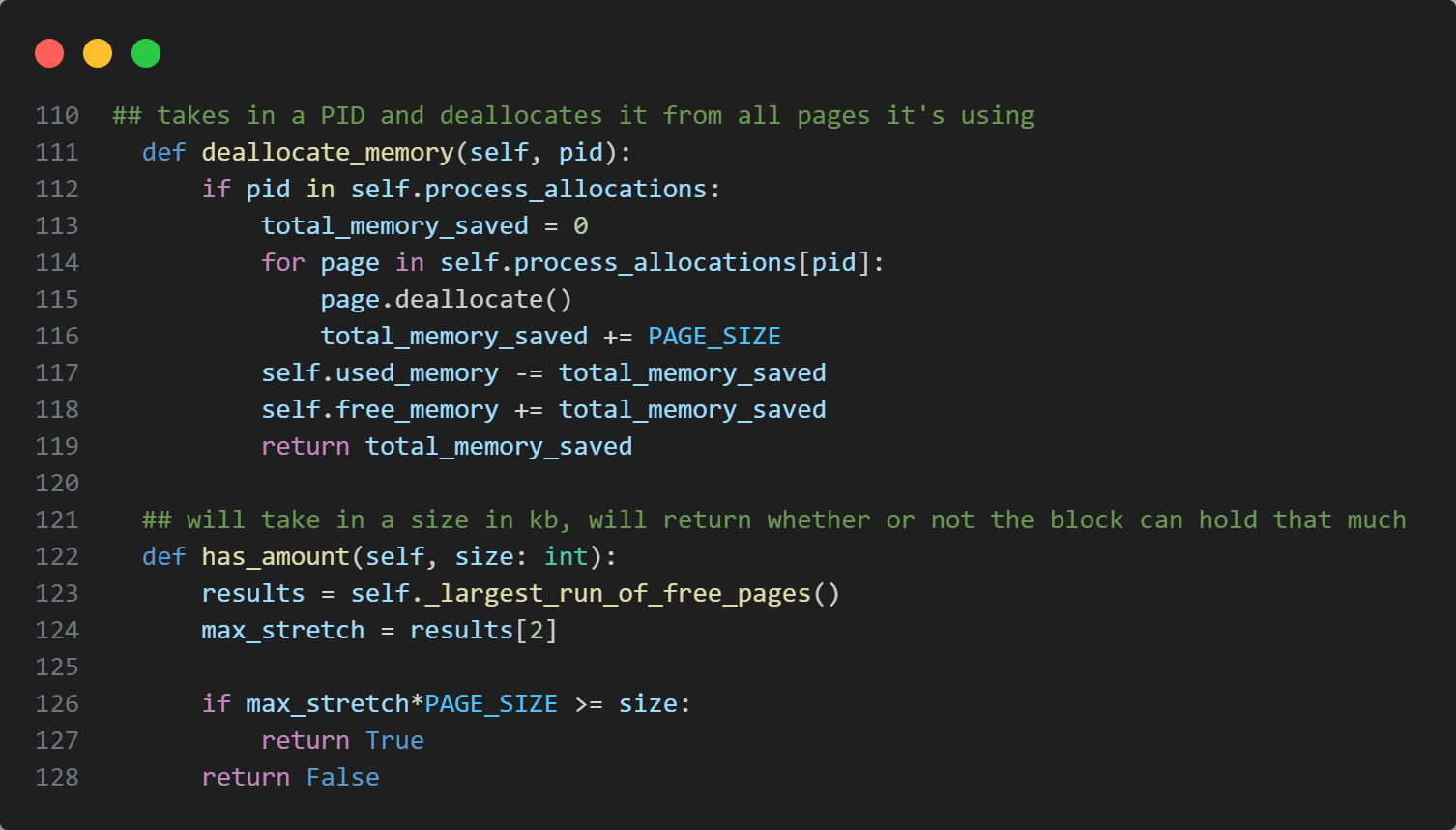


The Block class, and largest\_run method

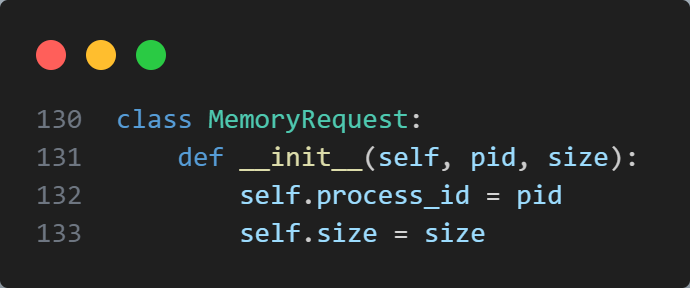


Memory Allocation, Deallocation and has\_amount functions

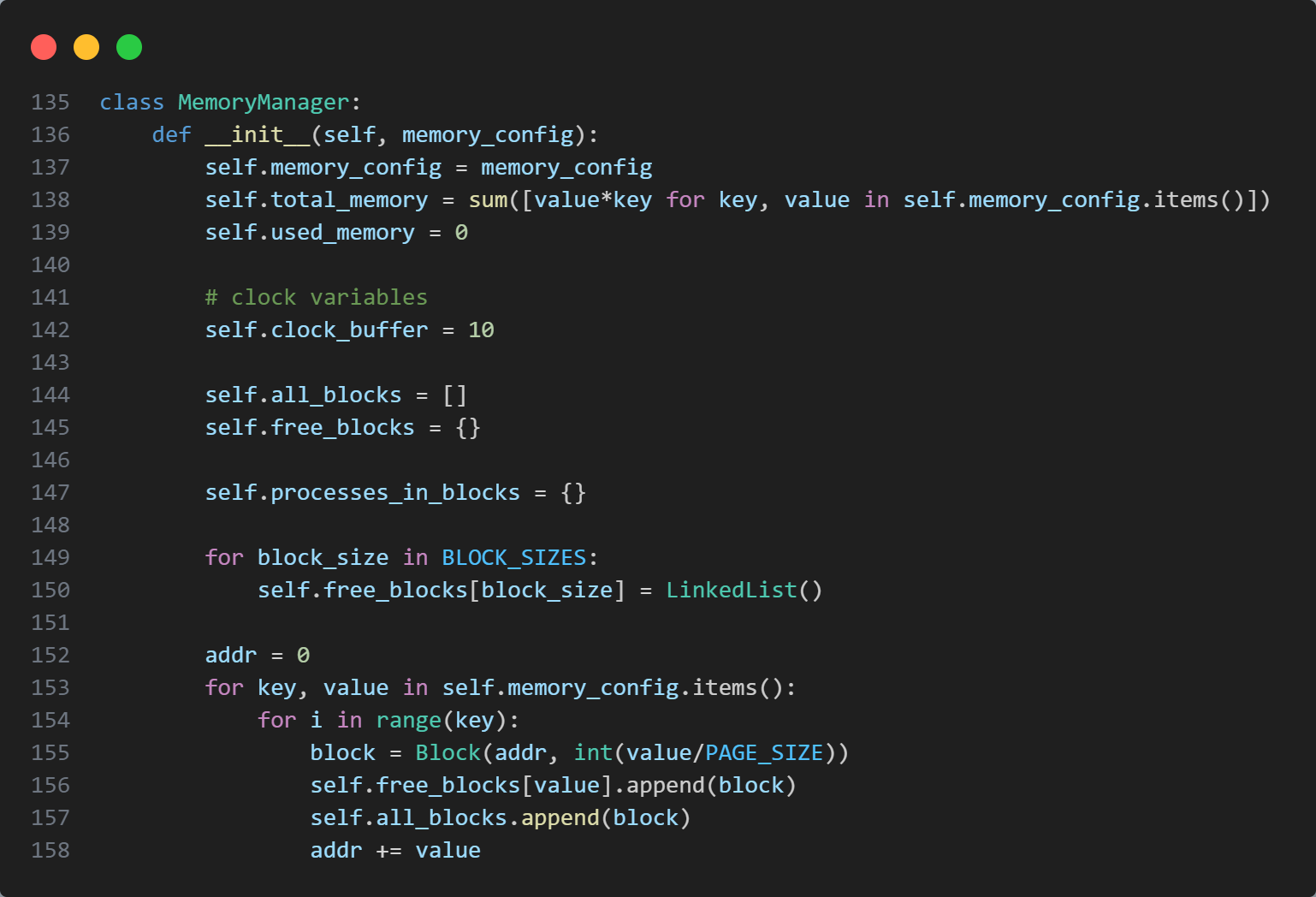




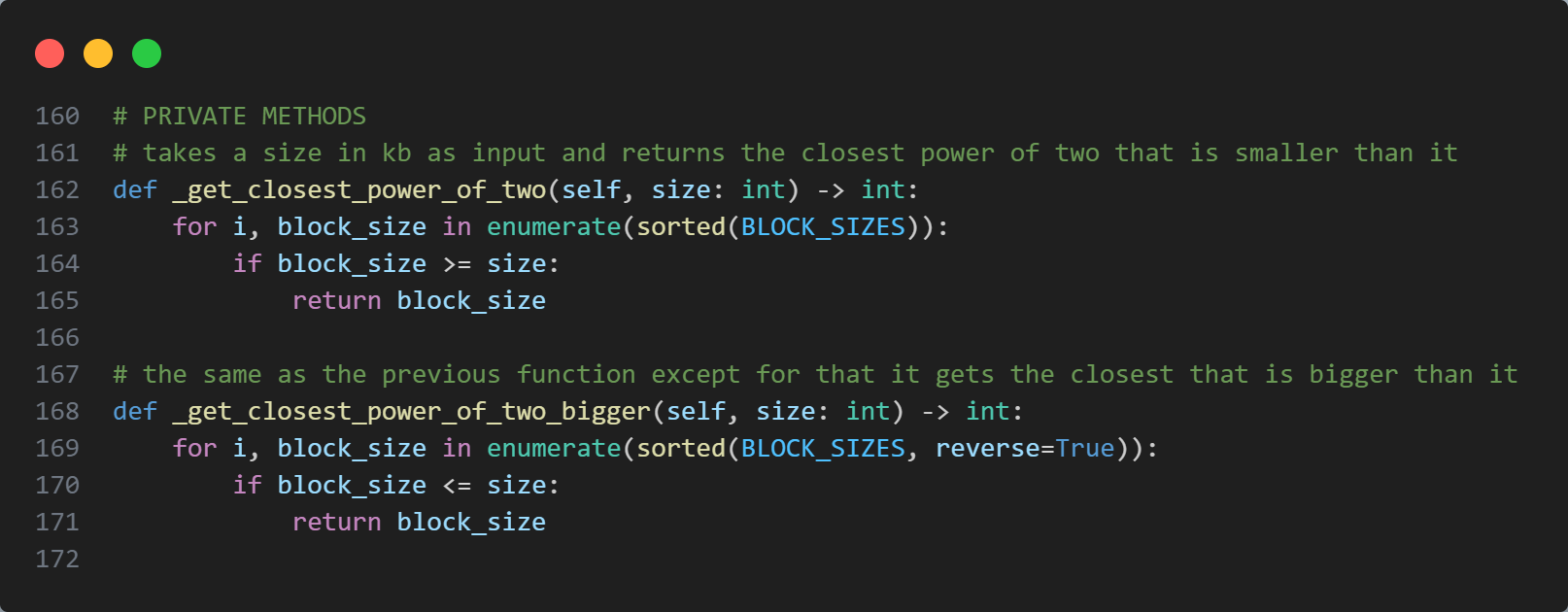
MemoryRequest class



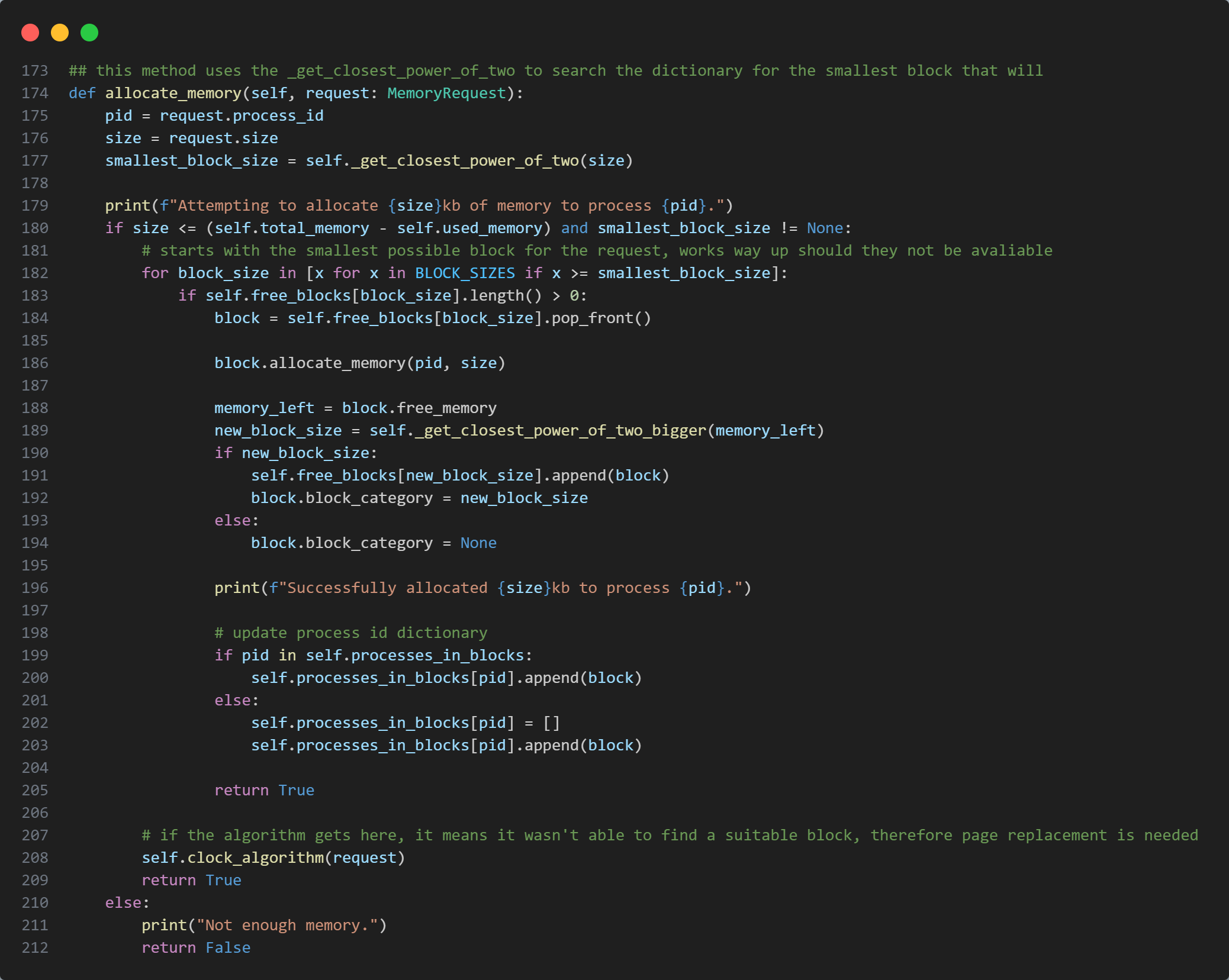
MemoryManager constructor



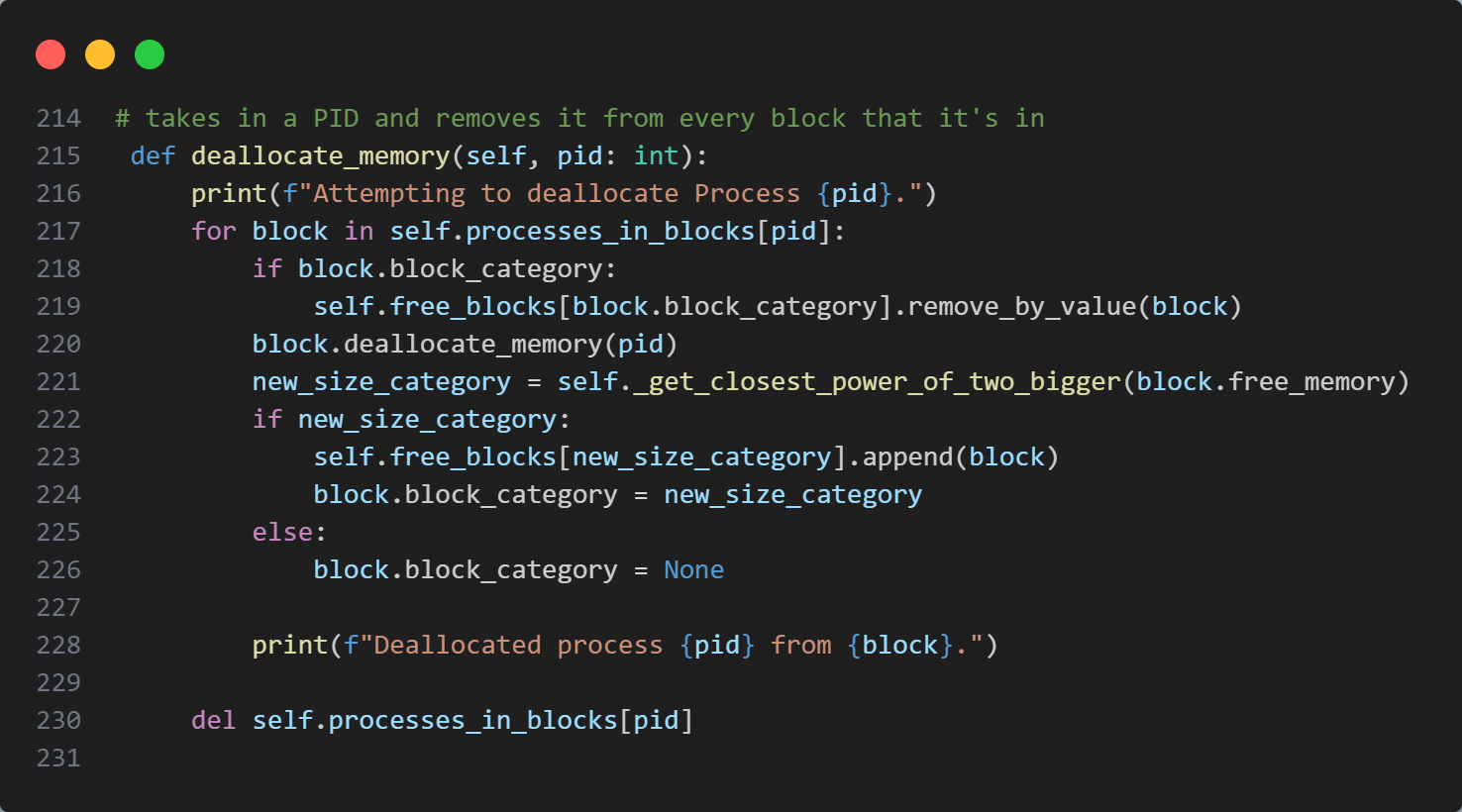
These functions are to help getting the correct block size.



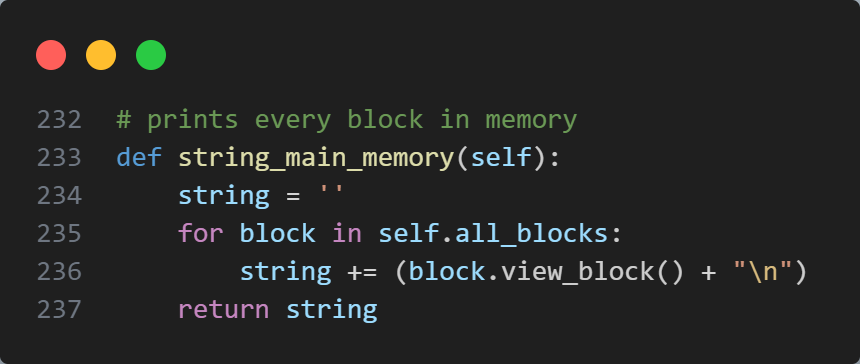
Allocate Memory function. Where best fit is used (or a modified version of it).



Deallocation Function



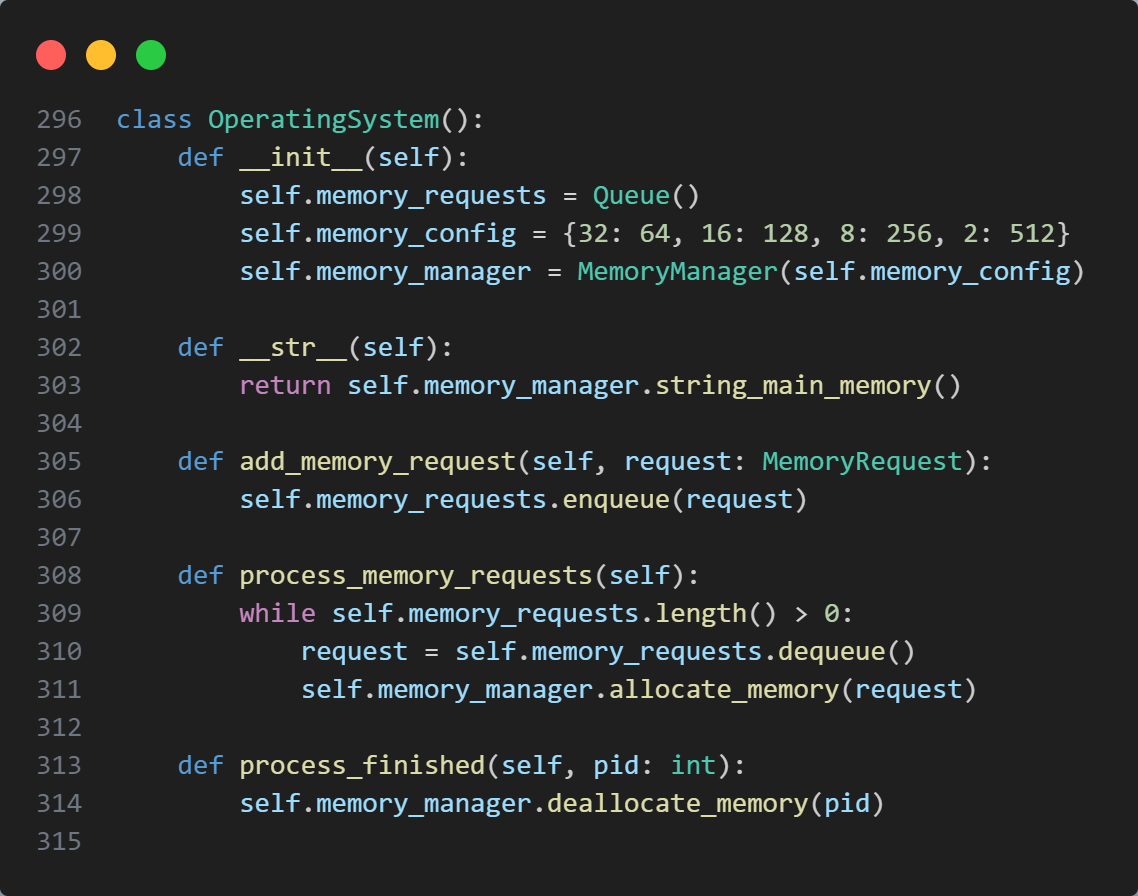
This function simply prints each block in memory



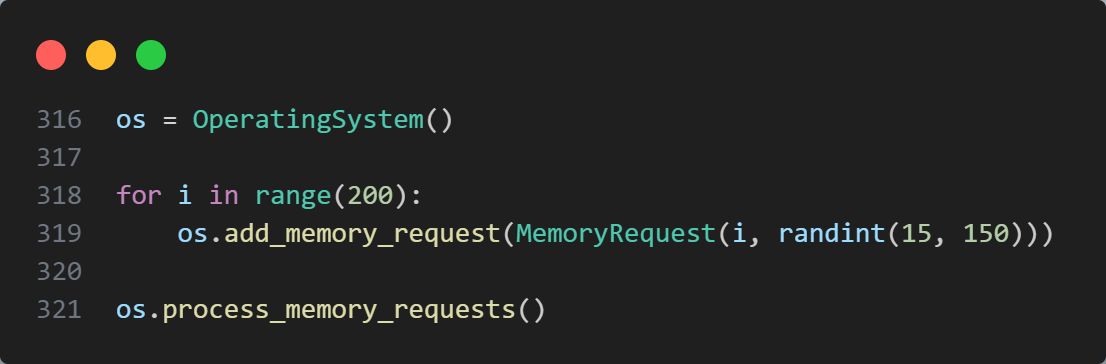
The clock algorithm function



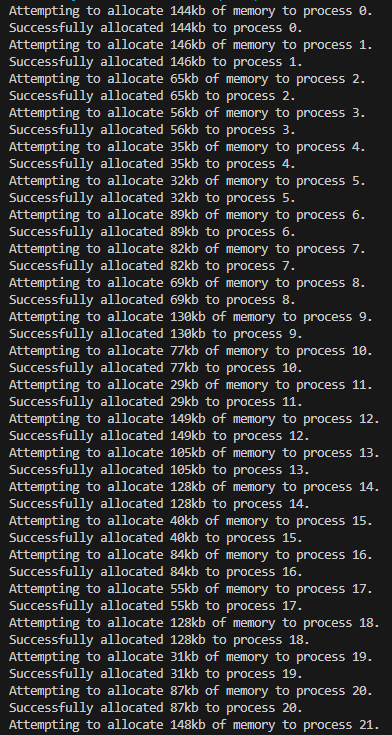
OperatingSystem class, this class simply wraps everything into a neat class and implement the FIFO Queue for memory requests.



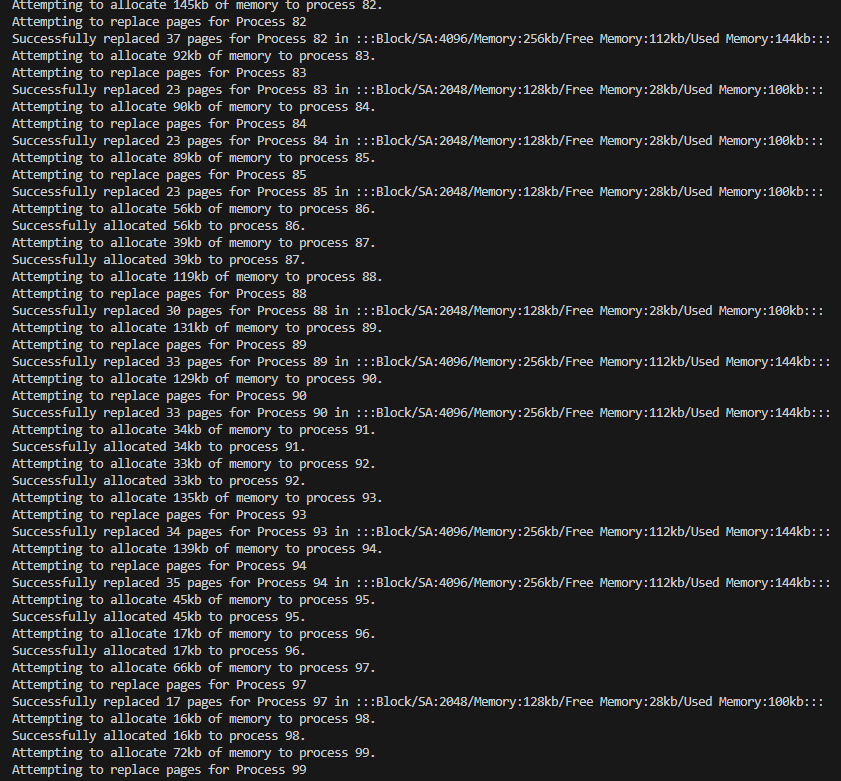
**Task 4**



This is what I will run and show the output of first. I create an OperatingSystem instance and add 200 memory requests of varying sizes to it.

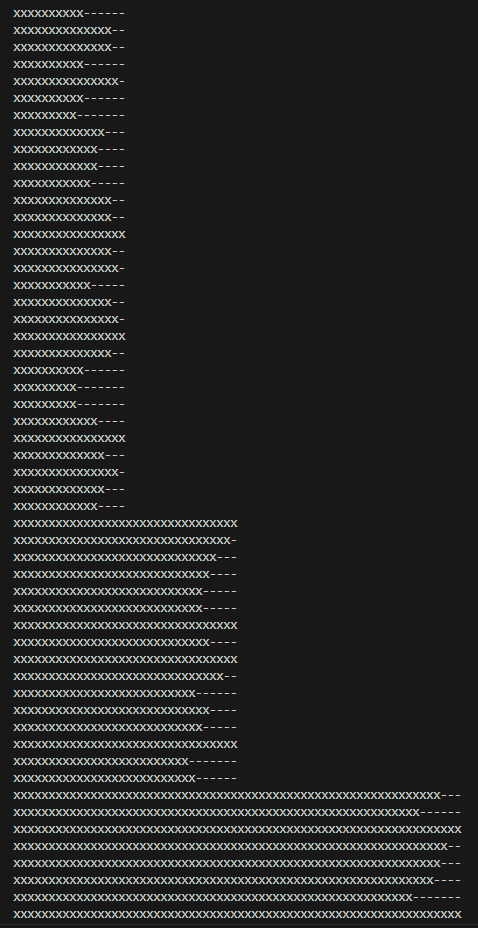


The output starts out normal, allocating memory to each process without the need for page replacement because the memory still has space left in it.



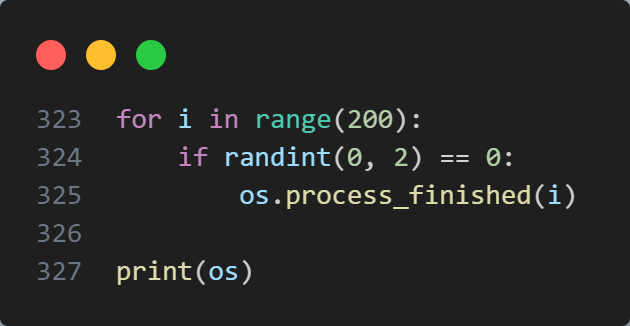
Later on, when the memory fills up, page replacement will be needed. You can see it here searching for new pages and allocating them to processes.

After all of the allocation, the string representation of all the blocks looks like this:



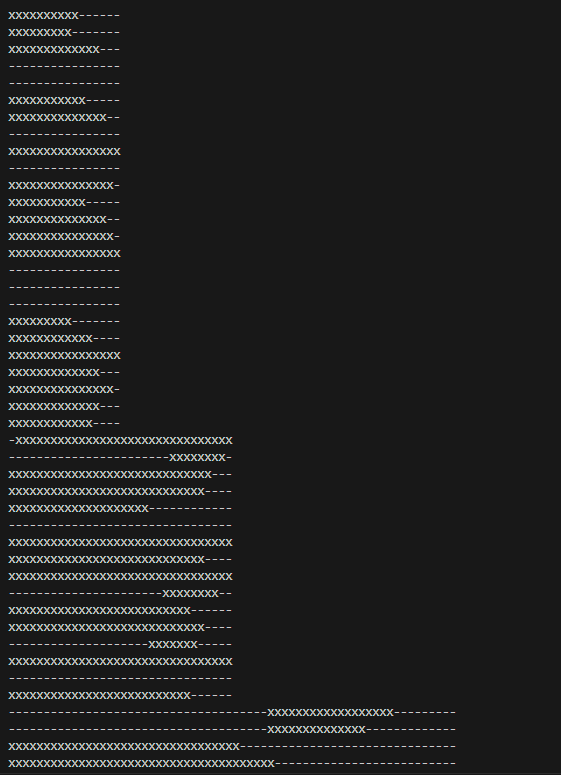
Where ‘x’ represents a allocated page, and ‘-‘ is an empty page, and each line represents a block.

For the second part of the simulation, I’ll run through each process and randomly deallocate certain ones.





It runs through and deallocates certain processes, which would simulate certain processes finishing their execution.

The string representation of the blocks now looks like this: 

This would simulate what a real system’s memory would look like as new processes are added and some are finished their execution.