import random

import time

# Simulate a memory pool (e.g., 50 units of memory for better console display)

MEMORY\_SIZE = 50

TIME\_STEPS = 15 # Number of time steps for simulation

# Class to represent a memory block

class MemoryBlock:

def \_\_init\_\_(self, start, size, process\_id, allocated\_time):

self.start = start # Starting address

self.size = size # Size of the block

self.process\_id = process\_id # ID of the process using this block

self.allocated\_time = allocated\_time # Time when block was allocated

self.freed\_time = None # Time when block was freed (None if still allocated)

# Class to manage memory and visualize allocation

class MemoryManager:

def \_\_init\_\_(self, memory\_size):

self.memory\_size = memory\_size

self.memory\_blocks = [] # List of allocated memory blocks

self.time = 0 # Current time step

def allocate(self, size, process\_id):

# Find a free space using First-Fit algorithm

if size > self.memory\_size:

print(f"Process {process\_id}: Requested size {size} is too large for memory!")

return False

# Sort blocks by start address to find gaps

self.memory\_blocks.sort(key=lambda x: x.start)

current\_pos = 0

# Look for a free space

for block in self.memory\_blocks:

if block.freed\_time is not None: # Skip freed blocks

continue

if current\_pos + size <= block.start: # Found a gap

new\_block = MemoryBlock(current\_pos, size, process\_id, self.time)

self.memory\_blocks.append(new\_block)

print(f"Process {process\_id}: Allocated {size} units at address {current\_pos}")

return True

current\_pos = block.start + block.size

# Check if there's space at the end of memory

if current\_pos + size <= self.memory\_size:

new\_block = MemoryBlock(current\_pos, size, process\_id, self.time)

self.memory\_blocks.append(new\_block)

print(f"Process {process\_id}: Allocated {size} units at address {current\_pos}")

return True

print(f"Process {process\_id}: No free space for {size} units!")

return False

def deallocate(self, process\_id):

# Deallocate memory for a given process

for block in self.memory\_blocks:

if block.process\_id == process\_id and block.freed\_time is None:

block.freed\_time = self.time

print(f"Process {process\_id}: Deallocated {block.size} units at address {block.start}")

return True

print(f"Process {process\_id}: No allocated memory found to deallocate!")

return False

def visualize(self):

# Create a text-based visualization of the memory state

memory\_display = ['-'] \* self.memory\_size # Initialize memory with free slots ('-')

# Fill in allocated blocks

for block in self.memory\_blocks:

if block.freed\_time is None or block.freed\_time > self.time: # Block is still allocated

for i in range(block.start, block.start + block.size):

memory\_display[i] = f"P{block.process\_id}"

# Print the memory state

print(f"\nTime Step {self.time}:")

print("Memory State: [", end="")

for i in range(self.memory\_size):

print(memory\_display[i], end="")

if (i + 1) % 10 == 0: # Add spacing for readability

print(" ", end="")

print("]")

print(f"Legend: '-' = Free, P<Number> = Process ID")

self.time += 1

# Simulate memory management

def simulate\_memory\_management():

manager = MemoryManager(MEMORY\_SIZE)

# Simulate allocation and deallocation over time

for t in range(TIME\_STEPS):

print(f"\n=== Time Step {t} ===")

# Randomly decide to allocate or deallocate

action = random.choice(["allocate", "deallocate", "none"])

if action == "allocate":

process\_id = random.randint(0, 4) # Random process ID (0 to 4)

size = random.randint(5, 10) # Random size between 5 and 10

manager.allocate(size, process\_id)

elif action == "deallocate":

process\_id = random.randint(0, 4) # Random process ID

manager.deallocate(process\_id)

# Visualize the memory state

manager.visualize()

time.sleep(1) # Add a delay to make the simulation easier to follow

# Run the simulation

if \_\_name\_\_ == "\_\_main\_\_":

simulate\_memory\_management()