Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра обчислювальної техніки

Лабораторна робота №2

з дисципліни
«Операційні системи»

Виконала:
студентка групи ІМ-21
Рабійчук Дар’я Олександрівна
номер в списку: 18

Перевірив:
Сімоненко А.В.

Київ 2024

Завдання

1) Реалізувати алокатор пам’яті загального призначення для програми користувача, що реалізує вище визначений API виконуючи вище зазначені умови та вимоги до алокатора пам’яті.

2) Для реалізації алокатора самостійно вибрати мову системного програмування.

3) Реалізувати функцію mem_show(), яка має виводити всі структури даних алокатора пам’яті.

4) Перевірити коректність реалізації алокатора пам’яті за допомогою автоматичного тестера.

5) Якщо є можливість, тоді використати розроблений алокатор пам’яті з реальною програмою, що використовує алокатор пам’яті. Якщо ця програма багатопотокова, тоді додати mutex до кожної функції API алокатора пам’яті.

Лістинг програми

Main.cs

```
using System;

using SlabAllocator;

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Запуск slab-алокатора пам'яті...");

        MemoryAllocator allocator = new MemoryAllocator();

        var ptr1 = allocator.Allocate(128);
        var ptr2 = allocator.Allocate(64);

        Console.WriteLine("Стан пам'яті після виділення:");
        allocator.MemShow();

        allocator.Deallocate(ptr1);
        allocator.Deallocate(ptr2);

        Console.WriteLine("Стан пам'яті після звільнення:");
        allocator.MemShow();

       Console.WriteLine("\n=== Автоматичне тестування алокатора пам'яті ===");
```

```
        // Виділення блоків пам'яті
        List<IntPtr> allocatedPointers = new List<IntPtr>();
        for (int i = 0; i < 10; i++)
        {
            IntPtr ptr = allocator.Allocate(128);
            if (ptr != IntPtr.Zero)
            {
                Console.WriteLine($"[ALLOC] Виділено {128} байт за адресою
{ptr}");
                allocatedPointers.Add(ptr);
            }
            else
            {
                Console.WriteLine("[ERROR] Помилка виділення пам'яті.");
            }
        }

                // Звільнення пам'яті
            foreach (var ptr in allocatedPointers)
        {
            allocator.Deallocate(ptr);
            Console.WriteLine($"[FREE] Звільнено пам'ять за адресою {ptr}");
        }

        // Видаляємо порожні Slabs
        allocator.MemShow(); // Показ стану до очищення
        allocator.RemoveEmptySlabs(); // Видаляємо порожні Slabs
        allocator.MemShow(); // Показ стану після очищення

    }

}
```

Slab.cs

```
using System;

using System.Collections;

namespace SlabAllocator
{
   public class Slab
{
    private readonly int objectSize;
    private readonly int objectCount;
    private readonly BitArray bitmap;
    private readonly byte[] memory;

    public Slab(int objectSize, int pageSize)
    {
        this.objectSize = objectSize;
```

```
    this.objectCount = pageSize / objectSize;
    this.bitmap = new BitArray(objectCount, true);
    this.memory = new byte[pageSize];
}

public IntPtr Allocate()
{
    for (int i = 0; i < objectCount; i++)
    {
        if (bitmap[i])
        {
            bitmap[i] = false; // Позначаємо блок як зайнятий
            return new IntPtr(i * objectSize);
        }
    }
    return IntPtr.Zero; // Усі блоки зайняті
}
    public void Deallocate(IntPtr ptr)
    {
        if (ptr == IntPtr.Zero) return;
        int index = ptr.ToInt32() / objectSize;
        if (index >= 0 && index < objectCount)
        {
            bitmap[index] = true; // Блок позначається як вільний
        }
    }


public bool IsEmpty()
{
    foreach (bool bit in bitmap)
    {
        if (bit) return true;
    }
    return false;
}

public int AllocatedSize => objectSize * objectCount;

public int UsedCount
{
    get
    {
        int count = 0;
        foreach (bool bit in bitmap)
        {
            if (!bit) count++; // Рахуємо зайняті блоки
        }
        return count;
    }
}
```

```
 }

}
```

MemoryAllocator.cs

```
using System;
using System.Collections.Generic;

namespace SlabAllocator
{
    public class MemoryAllocator
    {
        private const int PageSize = 4096;
        private Dictionary<int, List<Slab>> slabCache;

        public MemoryAllocator()
        {
            slabCache = new Dictionary<int, List<Slab>>();
        }

        public IntPtr Allocate(int size)
        {
            if (size <= 0) throw new ArgumentOutOfRangeException(nameof(size),
"Розмір має бути більше 0");

            int slabSize = GetNearestPowerOfTwo(size);

            if (!slabCache.ContainsKey(slabSize))
            {
                AddNewSlab(slabSize);
            }

            foreach (var slab in slabCache[slabSize])
            {
                IntPtr ptr = slab.Allocate();
                if (ptr != IntPtr.Zero)
                {
                    return ptr;
                }
            }

            Console.WriteLine($"[INFO] Усі наявні slabs для розміру {slabSize}
заповнені. Створюємо новий.");
            AddNewSlab(slabSize);
            return slabCache[slabSize][^1].Allocate();
        }

        public void Deallocate(IntPtr ptr)
        {
            if (ptr == IntPtr.Zero) return;
```

```
            foreach (var kvp in slabCache)
            {
                foreach (var slab in kvp.Value)
                {
                    slab.Deallocate(ptr);
                }
            }

            RemoveEmptySlabs();
        }

        public void MemShow()
        {
            Console.WriteLine("Стан пам'яті:");

            foreach (var kvp in slabCache)
            {
                int objectSize = kvp.Key;
                int slabCount = kvp.Value.Count;

                Console.WriteLine($"Кеш об'єктів розміром {objectSize} байт:
{slabCount} slabs.");
                foreach (var slab in kvp.Value)
                {
                    Console.WriteLine($"   Slab (Allocated: {slab.AllocatedSize},
Empty: {slab.IsEmpty()}, Used: {slab.UsedCount})");
                }
            }
        }

        private void AddNewSlab(int objectSize)
        {
            if (!slabCache.ContainsKey(objectSize))
            {
                slabCache[objectSize] = new List<Slab>();
            }
            slabCache[objectSize].Add(new Slab(objectSize, PageSize));
        }

        public void RemoveEmptySlabs()
        {
            foreach (var kvp in slabCache)
            {
                kvp.Value.RemoveAll(slab => slab.IsEmpty());
            }
        }

        private int GetNearestPowerOfTwo(int size)
        {
            int power = 1;
```

```
            while (power < size) power *= 2;
            return power;
        }
    }

}
```

MemoryCache.cs

```
using System;
using System.Collections.Generic;

namespace SlabAllocator
{
    public class MemoryCache
    {
        private readonly int objectSize;
        private readonly Queue<Slab> slabs;

        public MemoryCache(int objectSize, int pageSize)
        {
            this.objectSize = objectSize;
            this.slabs = new Queue<Slab>();
            AddNewSlab(pageSize);
        }

        public bool CanAllocate(int size)
        {
            return size <= objectSize;
        }

        public IntPtr Allocate()
        {
            foreach (var slab in slabs)
            {
                IntPtr ptr = slab.Allocate();
                if (ptr != IntPtr.Zero)
                {
                    return ptr;
                }
            }

            AddNewSlab(objectSize * 8);
            return slabs.Peek().Allocate();
        }

        public void Deallocate(IntPtr ptr)
        {
            foreach (var slab in slabs)
            {
```

```
                slab.Deallocate(ptr);
            }
        }
    public bool Contains(IntPtr ptr)
    {
        foreach (var slab in slabs)
        {
             if (!slab.IsEmpty() && ptr.ToInt32() >= 0 && ptr.ToInt32() <
slab.AllocatedSize)
            {
            return true;
            }
        }
        return false;
    }


        public void ShowStatus()
        {
            Console.WriteLine($"Кеш об'єктів розміром {objectSize} байт:
{slabs.Count} slabs.");
        }

        private void AddNewSlab(int pageSize)
        {
            slabs.Enqueue(new Slab(objectSize, pageSize));
        }


    }

}
```

AutoTetster.cs

```
using System;

using System.Collections.Generic;
using System.Security.Cryptography;
using System.Collections;

namespace SlabAllocator
{
    public class AutoTester
    {
        private class AllocatedBlock
        {
            public IntPtr Pointer { get; set; }
            public int Size { get; set; }
            public byte[] Data { get; set; } = Array.Empty<byte>();
            public byte[] Checksum { get; set; } = Array.Empty<byte>();
        }
```

```
        private Random random = new Random();
        private List<AllocatedBlock> allocatedBlocks = new
List<AllocatedBlock>();
        private MemoryAllocator allocator;

        public AutoTester(MemoryAllocator allocator)
        {
            this.allocator = allocator;
        }

        private byte[] ComputeChecksum(byte[] data)
        {
            using (SHA256 sha256 = SHA256.Create())
            {
                return sha256.ComputeHash(data);
            }
        }

        private byte[] FillRandomData(int size)
        {
            byte[] data = new byte[size];
            random.NextBytes(data);
            return data;
        }

        private bool VerifyBlock(AllocatedBlock block)
        {
            if (block == null || block.Pointer == IntPtr.Zero)
            {
                return false;
            }

            byte[] currentChecksum = ComputeChecksum(block.Data);
            return
StructuralComparisons.StructuralEqualityComparer.Equals(block.Checksum,
currentChecksum);
        }

        public void RunTests(int iterations)
        {
            Console.WriteLine("\n[INFO] Running automated tests...");

            for (int i = 0; i < iterations; i++)
            {
                int action = random.Next(3);
                switch (action)
                {
                    case 0: PerformAlloc(); break;
                    case 1: PerformRealloc(); break;
                    case 2: PerformFree(); break;
```

```
                }
            }

            Cleanup();
            Console.WriteLine("[INFO] Testing completed.");
        }

        private void PerformAlloc()
        {
            int size = random.Next(16, 512);
            IntPtr pointer = allocator.Allocate(size);


            if (pointer != IntPtr.Zero)
            {
                byte[] data = FillRandomData(size);
                byte[] checksum = ComputeChecksum(data);

                allocatedBlocks.Add(new AllocatedBlock
                {
                    Pointer = pointer,
                    Size = size,
                    Data = data,
                    Checksum = checksum
                });

                Console.WriteLine($"[ALLOC] Allocated {size} bytes.");
            }
            else
            {
                Console.WriteLine("[ERROR] Allocate failed to provide memory.");
            }
        }

                private void PerformRealloc()
        {
            if (allocatedBlocks.Count == 0) return;

            int index = random.Next(allocatedBlocks.Count);
            AllocatedBlock block = allocatedBlocks[index];

            if (!VerifyBlock(block))
            {
                Console.WriteLine("[ERROR] Data corruption detected before
realloc.");
                return;
            }

            int newSize = random.Next(16, 2048);
            IntPtr newPointer = allocator.Allocate(newSize);
```

```
            if (newPointer == IntPtr.Zero)
            {
                Console.WriteLine("[ERROR] Failed to allocate memory for
realloc.");
                return;
            }

            try
            {
                if (block.Data == null || block.Pointer == IntPtr.Zero)
                {
                    Console.WriteLine("[ERROR] Null reference during realloc!");
                    allocator.Deallocate(newPointer); // Звільняємо новий блок
                    return;
                }

                unsafe
                {
                    int copySize = Math.Min(block.Size, newSize);
                    if (copySize > 0)
                    {
                        Buffer.MemoryCopy((void*)block.Pointer,
(void*)newPointer, newSize, copySize);
                    }
                }

                byte[] newData = new byte[newSize];
                Array.Copy(block.Data, newData, Math.Min(block.Size, newSize));
                block.Data = newData;
                block.Checksum = ComputeChecksum(newData);
            }
            catch (Exception ex)
            {
                Console.WriteLine($"[ERROR] Memory copy failed: {ex.Message}");
                allocator.Deallocate(newPointer);
                return;
            }

            allocator.Deallocate(block.Pointer);

            block.Pointer = newPointer;
            block.Size = newSize;

            Console.WriteLine($"[REALLOC] Block resized to {newSize} bytes.");
        }


        private void PerformFree()
        {
            if (allocatedBlocks.Count == 0) return;
```

```
            int index = random.Next(allocatedBlocks.Count);
            AllocatedBlock block = allocatedBlocks[index];

            if (!VerifyBlock(block))
            {
                Console.WriteLine("[ERROR] Data corruption detected before
free!");
                return;
            }

            allocator.Deallocate(block.Pointer);
            allocatedBlocks.RemoveAt(index);
            Console.WriteLine("[FREE] Block deallocated.");
        }

        private void Cleanup()
        {
            foreach (var block in allocatedBlocks)
            {
                if (!VerifyBlock(block))
                {
                    Console.WriteLine("[ERROR] Data corruption detected during
cleanup!");
                }
                allocator.Deallocate(block.Pointer);
            }

            allocatedBlocks.Clear();
            Console.WriteLine("[INFO] All blocks deallocated.");
        }
    }

    public class Extent
    {
        public int Size { get; }
        public bool IsFree { get; private set; }

        public Extent(int size)
        {
            Size = size;
            IsFree = true;
        }

        public void MarkUsed()
        {
            IsFree = false;
        }

        public void MarkFree()
        {
            IsFree = true;
```

```
        }
    }
}
```

Приклад виводу у консолі

```
Запуск slab-алокатора пам'яті...
[INFO] Усі наявні slabs для розміру 128 заповнені. Створюємо новий.
[INFO] Усі наявні slabs для розміру 64 заповнені. Створюємо новий.
Стан пам'яті після виділення:
Стан пам'яті:
Кеш об'єктів розміром 128 байт: 2 slabs.
   Slab (Allocated: 4096, Empty: True, Used: 1)
   Slab (Allocated: 4096, Empty: True, Used: 1)
Кеш об'єктів розміром 64 байт: 2 slabs.
   Slab (Allocated: 4096, Empty: True, Used: 1)
   Slab (Allocated: 4096, Empty: True, Used: 1)
Стан пам'яті після звільнення:
Стан пам'яті:
Кеш об'єктів розміром 128 байт: 2 slabs.
   Slab (Allocated: 4096, Empty: True, Used: 1)
   Slab (Allocated: 4096, Empty: True, Used: 1)
Кеш об'єктів розміром 64 байт: 2 slabs.
   Slab (Allocated: 4096, Empty: True, Used: 1)
   Slab (Allocated: 4096, Empty: True, Used: 1)

=== Автоматичне тестування алокатора пам'яті ===
[ALLOC] Виділено 128 байт за адресою 128
[ALLOC] Виділено 128 байт за адресою 256
[ALLOC] Виділено 128 байт за адресою 384
[ALLOC] Виділено 128 байт за адресою 512
[ALLOC] Виділено 128 байт за адресою 640
[ALLOC] Виділено 128 байт за адресою 768
[ALLOC] Виділено 128 байт за адресою 896
[ALLOC] Виділено 128 байт за адресою 1024
```

```
[ALLOC] Виділено 128 байт за адресою 1152
[ALLOC] Виділено 128 байт за адресою 1280
[FREE] Звільнено пам'ять за адресою 128
[FREE] Звільнено пам'ять за адресою 256
[FREE] Звільнено пам'ять за адресою 384
[FREE] Звільнено пам'ять за адресою 512
[FREE] Звільнено пам'ять за адресою 640
[FREE] Звільнено пам'ять за адресою 768
[FREE] Звільнено пам'ять за адресою 896
[FREE] Звільнено пам'ять за адресою 1024
[FREE] Звільнено пам'ять за адресою 1152
[FREE] Звільнено пам'ять за адресою 1280
Стан пам'яті:
Кеш об'єктів розміром 128 байт: 0 slabs.
Кеш об'єктів розміром 64 байт: 0 slabs.
Стан пам'яті:
Кеш об'єктів розміром 128 байт: 0 slabs.
Кеш об'єктів розміром 64 байт: 0 slabs.
```