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 №3

«Алгоритми заміни сторінок»

з дисципліни
«Операційні системи»

Виконала:

студентка групи ІМ-11
Бащак Ярина Володимирівна

Перевірив:

ст. вик. Сімоненко А.В.

Київ 2023

Завдання.

Завдання на роботу

- Розробити модель (програму) системи зі сторінкової організацією пам'яті, що відповідає наведеним вище початковим даними, з реалізацією двох алгоритмів заміни сторінок відповідно до варіанту.
- Програма має в стандартний потік виведення (або у файл) виводити звіт про характеристики віртуальної пам'яті процесів, сторінкові промахи, рішення алгоритму заміни сторінок і т. ін.
- Алгоритми заміни сторінок вибираються за варіантом (номер варіанту визначається за номером залікової книжки, останні_дві_цифри mod 3 + 1).
- Шляхом зміни розміру робочого набору перевірити правильність реалізації моделі та реалізованого алгоритму заміни сторінок. Якщо кількість сторінок у робочому наборі збільшувати, тоді має збільшуватись частота сторінкових промахів. Алгоритм випадкової заміни сторінок має призводити до більшої частоти сторінкових промахів ніж інший алгоритм з варіанту.
- Бажано в програмі виокремити наступні класи (структури): процес, ядро, MMU, фізична сторінка, таблиця сторінок, робочий набір.

Опис алгоритму NRU.

Я використовувала мову C# (.NET). Для реалізації цього алгоритму я створила окремий клас NRUAlgorithm, екзампляр якого створюється в класі MMU.
Всі зайняті сторінки зберігаються в екземлярі NRUAlgorithm в полі ClassifiedPages, який є масивом з 4-х елементів для кожного класу сторінок (00, 01, 10, 11).

```
public readonly List<PhysicalPage>[] ClassifiedPages = new List<PhysicalPage>[4];
```

Цей масив заповнюватися значеннями тоді, коли процеси звертаються до своїх віртуальних сторінок, які на даний момент не відображені.

Це видно у методі AccessPage класу MMU (останній умовний оператор):

```
public void AccessPage(VirtualPage[] pageTable, int idx, bool isModified, Kernel kernel)
    {
        PhysicalPage? physicalPage = default;

        if (!pageTable[idx].P)
        {
            physicalPage = kernel.PageFault(pageTable, idx);
        }

        pageTable[idx].R = true;

        if (isModified) pageTable[idx].M = true;

        // only for NRU Algolithm
        if (physicalPage != null)
        {
            NRUAlgorithm.AddPageToAppropriateClass(physicalPage);
```

```
        }
    }
```

Метод AddPageToApproprialeClass виглядає так:

```
public void AddPageToAppropriateClass(PhysicalPage page)
    {
        int index = CalculatePageClass(page);
        ClassifiedPages[index].Add(page);
    }

public int CalculatePageClass(PhysicalPage page)
    {
        var virtualPage = page.PageTable![page.Idx];

        int index = 0;
        if (virtualPage.R) index += 2;
        if (virtualPage.M) index += 1;

        return index;
    }
```

Цей метод виконує тільки додавання сторінки в кінець відповідного класу, бо як тільки метод PageFault класу Kernel обрав фізичну сторінку, яку можна замінити, він видалив її з попереднього класу.

Робота самого алгоритму виглядає так:

```
public PhysicalPage NRUReplacementAlgorithm()
    {
        for (int i = 0; i < ClassifiedPages.Length; i++)
        {
            for (int j = 0; j < ClassifiedPages[i].Count; j++)
            {
                var physicalPage = ClassifiedPages[i][j];

                ClassifiedPages[i].RemoveAt(j);
                j--;

                if (CalculatePageClass(physicalPage) == i)
                {
                    return physicalPage;
                }
                else
                {
                    AddPageToAppropriateClass(physicalPage);
                }
            }
        }

        throw new Exception("No pages to replace!");
    }
```

Тут я йду по черзі по кожному списку (в порядку класів 00, 01, 10, 11) доти, доки не знайду першу сторінку, яка відповідає списку, в якому знаходиться. Як видно з рядків:

```
ClassifiedPages[i].RemoveAt(j);
j--;
```

кожну сторінку, яку я переглядаю, я одразу видаляю зі списку. Це тому що, якщо ця сторінка знаходиться в своєму списку, то я її видаляю і повертаю (потім ця ж сторінка буде додана в кінець відповідного класу, як я писала раніше), а якщо не у своєму, то її все одно потрібно видалити і додати до відповідного списку. Таким чином ми не «сортуємо і вибираємо» сторінки, а «вибираємо і сортуємо».

Але це не все, тому що інформацію про сторінки варто іноді оновлювати. За це відповідають два методи у класі NRUAlgorithm: ClearAllReferenceBits() і UpdateNPages(int n).

1) Метод ClearAllReferenceBits() скидає всі біти R, переміщаючи сторінки у відповідні класи (00 або 01).

```
    public void ClearAllReferenceBits()
    {
        for (int i = 0; i < ClassifiedPages.Length; i++)
        {
            for (int j = 0; j < ClassifiedPages[i].Count; j++)
            {
                var physicalPage = ClassifiedPages[i][j];
                if (physicalPage.PageTable![physicalPage.Idx].R)
                {
                    physicalPage.PageTable![physicalPage.Idx].R = false;
                    ClassifiedPages[i].RemoveAt(j);
                    j--;
                    AddPageToAppropriateClass(physicalPage);
                }
            }
        }

        Console.ForegroundColor = ConsoleColor.DarkRed;
        Console.WriteLine($"All reference bits were cleared.");
        Console.ResetColor();
    }
```

Цей метод запускається через кожен квант часу. Розмір кванту часу задається на при створенні ядра, наприклад, 500 запитів до пам’яті.

```
        int quantumOfTime = 500;
        var kernel = new Kernel(…, quantumOfTime, …);
```

2) Метод UpdateNPages(int n) проходить по частині сторінок (n сторінок, починаючи з першого класу) і якщо клас сторінки не співпадає з тим, в якому вона зараз знаходиться – переміщає її в потрібний.

```
public void UpdateNPages(int n)
{
    int count = 0;
    int i = 0;
    int j = 0;
    int numberOfUpdatedPages = 0;

    while (count < n)
    {
        if (ClassifiedPages[i].Count == 0)
        {
            i++;
            continue;
        }

        if (j >= ClassifiedPages[i].Count)
        {
            i++;
            j = 0;
        }

        var physicalPage = ClassifiedPages[i][j];

        if (CalculatePageClass(physicalPage) != i)
        {
            ClassifiedPages[i].RemoveAt(j);
            j--;
            AddPageToAppropriateClass(physicalPage);
            numberOfUpdatedPages++;
        }

        j++;
        count++;
    }

    Console.ForegroundColor = ConsoleColor.Green;
    Console.WriteLine($"{numberOfUpdatedPages} pages was updated.");
    Console.ResetColor();
}
```

Також метод відслідковує скільки сторінок з заданого числа n було оновлено і виводить цю інформацію в консоль. Цей метод запускається через кожен інтервал IntervalToUpdateSomePages, який вимірюється в кількості запитів і перевіряє NumberOfPagesToUpdateEachInterval сторінок. Ці значення також задаються при створенні ядра. Наприклад, через кожні 150 запитів перевіряти 20 сторінок.

Оскільки при скиданні всіх бітів R відбувається перенесення всіх змінених сторінок до відповідних класів, якщо час виклику ClearAllReferenceBits() (наприклад, кожні 500 запитів) співпадає по часу з викликом UpdateNPages(int n) (кожні 150 запитів), то після першого методу всі сторінки вже знаходитимуться у своїх класах і викликати останній метод немає сенсу.

Це відтворено в моїй програмі таким чином (ця перевірка відбувається після кожного виконаного сету запитів одиним процесом = кожні 40-60 запитів):

```
            if (IsItNewQuantumOfTime())
            {
                MemoryManager.NRUAlgorithm.ClearAllReferenceBits();
                IsTimeToUpdateSomePages();
            }
            else if (IsTimeToUpdateSomePages())
            {
                MemoryManager.NRUAlgorithm.UpdateNPages(NumberOfPagesToUpdateEachIn
terval);
            }
```

Опис використаних методів:

```
    private bool IsTimeToUpdateSomePages()
    {
        if (NumberOfRequestsFromLastUpdate >= IntervalToUpdateSomePages)
        {
            NumberOfRequestsFromLastUpdate -= IntervalToUpdateSomePages;
            return true;
        }
        return false;
    }

    private bool IsItNewQuantumOfTime()
    {
        if (NumberOfRequestsWithThisQuantumOfTime >= QuantumOfTime)
        {
            NumberOfRequestsWithThisQuantumOfTime -= QuantumOfTime;
            return true;
        }
        return false;
    }
```

Початкові дані.

Початкові дані, з якими створюється ядро:

```
        int numberOfPhysicalPages = 100;
        int maxProcessCount = 10;
        uint startPageNumber = 0x00010000;
        int startProcessCount = 4;
        int quantumOfTime = 500;
        int workingSetPercentage = 50; // specifies size of working set as
percentage of total number of virtual pages of process
        int intervalToGenerateNewWorkingSet = 200;
        int intervalToUpdateSomePages = 150;
        int numberOfPagesToUpdateEachInterval = 20;

        var kernel = new Kernel(
            maxProcessCount,
            numberOfPhysicalPages,
            startPageNumber,
            startProcessCount,
            quantumOfTime,
            workingSetPercentage,
            intervalToGenerateNewWorkingSet,
            intervalToUpdateSomePages,
            numberOfPagesToUpdateEachInterval
        );
```

Решта даних, які визначаються рандомним числом з певного діапазону:

- характеристики процесу: кількість віртуальних сторінок і необхідна кількість зроблених запитів для його завершення:

```
        var numberOfVirtualPages = Rand.Next(30, 60);
        var requiredNumberOfRequests = Rand.Next(300, 600);
```

- кількість запитів, які виконає процес, коли до нього (знову) дійшла черга, визначається кожен раз новим числом:

```
        int numberOfRequests = Rand.Next(40, 60);
```

- співвідношення запитів до сторінок з робочого набору до всіх сторінок 90/10:

```
        if (Rand.Next(100) < 90)
```

- співвідношення модифікації сторінки до лише читання 30/70:

```
        bool isModified = Rand.Next(10) <= 2; // modify/not-modify = 30/70
```

Що виводиться у консоль.

Певні методи виводять інформацію про результати своєї роботи у консоль, щоб відстежувати ключові моменти роботи програми. Це інформація про:

- створення ядра

```
Kernel with 100 number of physical pages, 4 start processes, 10 ma
x process count, 500 quantum of time was created.
```

- створення нового процесу

```
New process with id 6, 41 virtual pages, 20 size of working set,
386 required number of requests was created.
```

- виконання процесом певної кількості запитів разом з числом сторінкових промахів, які відбулися

```
Process with id 2 has made 48 requests, 21 of which were page faults.
```

- статистику вільних і зайнятих фізичних сторінок (виводиться тільки тоді, коли кількість вільних сторінок не нуль, бо інакше це займає надто багато рядків у консолі, що заважає відслідковувати роботу програми)

```
Process with id 1 has made 52 requests, 17 of which were page faults.
Busy pages: 17.
Free pages: 83.
Process with id 2 has made 48 requests, 21 of which were page faults.
Busy pages: 38.
Free pages: 62.
Process with id 3 has made 48 requests, 20 of which were page faults.
Busy pages: 58.
Free pages: 42.
Process with id 4 has made 47 requests, 21 of which were page faults.
Busy pages: 79.
Free pages: 21.
```

- створення нового робочого набору для певного процесу

```
New working set for process 1 was generated: 26, 1, 14, 11, 16, 27, 32,
15, 9, 4, 7, 5, 30, 23, 17, 12
```

- скидання всіх бітів R (виконання методу ClearAllReferenceBits()) і оновлення частини сторінок (виконання методу UpdateNPages(int n))

```
All reference bits were cleared.
Process with id 6 has made 58 requests, 17 of which were page faults.
Process with id 7 has made 58 requests, 8 of which were page faults.
3 pages was updated.
Process with id 8 has made 40 requests, 15 of which were page faults.
Process with id 9 has made 52 requests, 16 of which were page faults.
Process with id 10 has made 50 requests, 22 of which were page faults.
4 pages was updated.
Process with id 6 has made 49 requests, 13 of which were page faults.
```

- завершення процесу

```
Finished: process with id 6 was completed and removed from queue.
```

- загальна кількість зроблених запитів, загальна кількість сторінкових промахів, відсоток сторінкових промахів із загальної кількості запитів (після завершення всіх процесів)

```
Total requests: 5046.
Total page faults: 1386.
27% of requests were page faults.
```

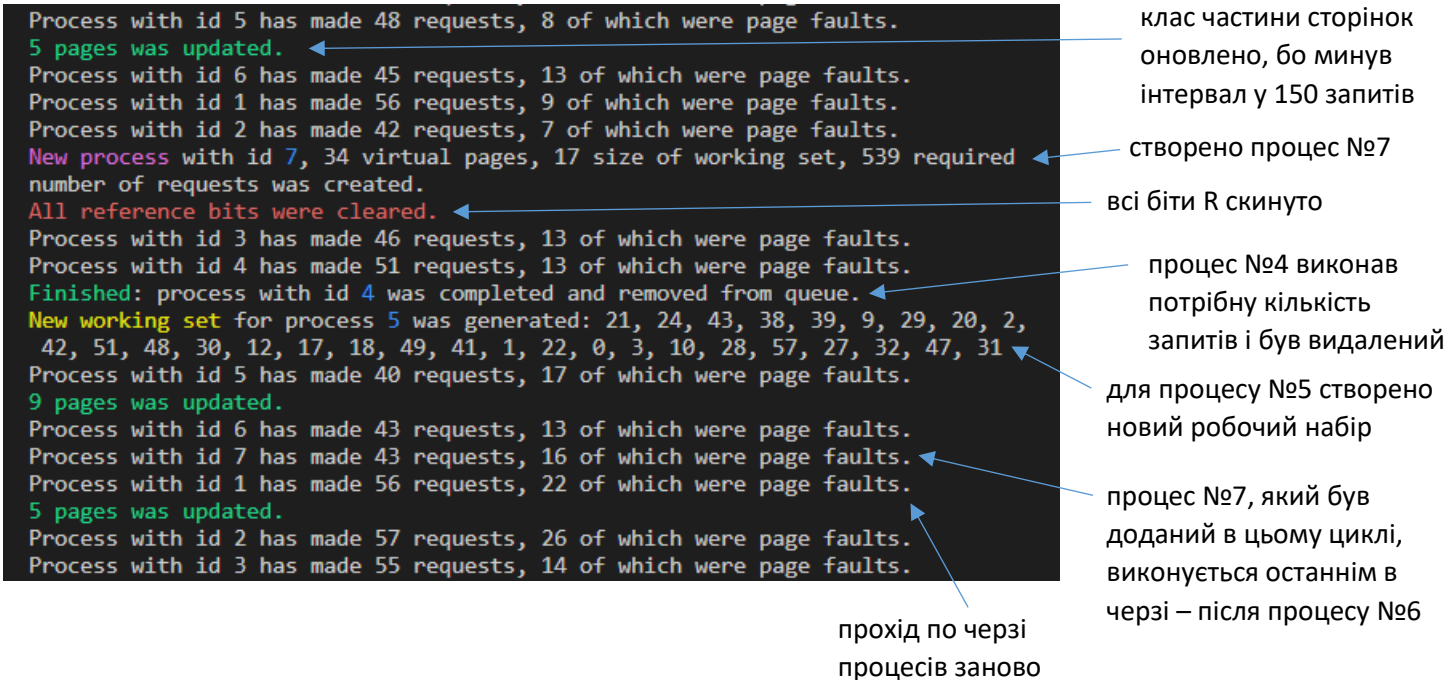
- можна також виводити інформацію про кожен сторінковий промах, якщо розкоментувати відповідний рядок в коді

```
// Console.WriteLine($"Page Fault. Number of page to replace:
{pageToReplace.PPN:X8}");
```

але наразі цей вивід відключений, бо це, як видно по кількості сторінкових промахів, приблизно 1400 додаткових рядків у консолі, що просто в неї не поміщається. Тому на заміну цьому кожен процес, коли завершує виконання чергового сету запитів, вказує також кількість сторінокових промахів, що відбулися. За бажанням вивід кожного промаху можна знову включити

```
Page Fault. Number of page to replace: 00010029
Page Fault. Number of page to replace: 00010013
Page Fault. Number of page to replace: 00010043
Page Fault. Number of page to replace: 00010040
Page Fault. Number of page to replace: 0001005C
Page Fault. Number of page to replace: 0001000C
Page Fault. Number of page to replace: 0001002E
Page Fault. Number of page to replace: 0001004C
Process with id 10 has made 54 requests, 8 of which were page faults.
```

Приблизний загальний вигляд консолі при виконанні програми:



Порівняння алгоритмів.

Оскільки частина параметрів для кожного нового запуску інша, яка задається рандомним чином, то відсоток сторінкових промахів постійно різний, але все таки коливаться в певному діапазоні.

Щоб могти чітко порівняти дію алгоритмів випадкової заміни і NRU, я створила таблиці, де записувала результати запусків програми, з одиними і тими ж вхідними параметрами по 10 разів і обчислювала середній відсоток сторінкових промахів.

working set = 50%		
	Random Replacement	NRU Algorithm
1	27	25
2	28	30
3	25	23
4	30	21
5	29	31
6	26	30
7	34	22
8	33	29
9	30	22
10	30	22
середнє	29,2	25,5

working set = 20%		
	Random Replacement	NRU Algorithm
1	16	9
2	13	10
3	13	9
4	11	11
5	12	9
6	11	9
7	13	10
8	13	9
9	12	9
10	13	9
середнє	12,7	9,4

По результатах видно, що алгоритм NRU все ж призводить до меншої кількості сторінкових промахів в порівнянні з випадковою заміною. Теж саме відбувається і при зменшенні робочого набору.

Висновок: в ході цієї лабораторної роботи я більше дізналася про алгоритми заміни сторінок і спробувала сама змоделювати їх роботу. По результатах видно, що програма на основі алгоритму NRU працює краще ніж на випадковій заміні. Різниця, правда, не суттєва, але є. Ще думаю, що ця різниця дуже залежить від підбору вхідних параметрів ядра і можна знайти такі вхідні набори, що різниця між цими двома алгоритмами буде більш помітною. Також було показано, що зменшення робочого набору призводить до зменшення відсотку сторінкових промахів.

Лістинг.

В цьому звіті були наведені тільки деякі фрагманти програми. Повний код пропоную нижче або за посиланням на github.

Важливо: даний варіант коду зараз працює за алгоритмом NRU. Якщо ви хочете перейти до алгоритму випадкової заміни, потрібно знайти всі коментарі

```
// only for NRU Algolithm
```

і закоментувати весь код, що до нього відноситься, а потім знайти всі

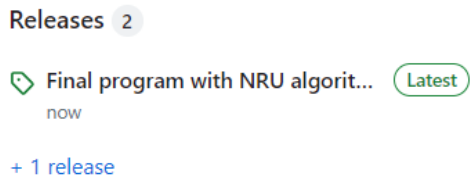
```
// only for Random Replacement Algolithm
```

і відповідно розкоментувати.

Наприклад, як в цих рядках

```
physicalPage = RandomReplacementAlgorithm(); // only for Random Replacement Algolithm
// physicalPage = MemoryManager.NRUAlgorithm.NRUReplacementAlgorithm(); // only for NRU Algolithm
```

Або для цього також створені два релізи на github:



Код класу Program і решти класів, які реалізують потрібну поведінку:

```
namespace lab1;

class Program
{
    static void Main(string[] args)
    {
        int numberOfPhysicalPages = 100;
        int maxProcessCount = 10;
        uint startPageNumber = 0x00010000;
        int startProcessCount = 4;
        int quantumOfTime = 500;
        int workingSetPercentage = 50; // specifies size of working set as percentage of total number of virtual pages of process
        int intervalToGenerateNewWorkingSet = 200;
        int intervalToUpdateSomePages = 150;
        int numberOfPagesToUpdateEachInterval = 20;

        var kernel = new Kernel(
            maxProcessCount,
            numberOfPhysicalPages,
            startPageNumber,
            startProcessCount,
            quantumOfTime,
            workingSetPercentage,
            intervalToGenerateNewWorkingSet,
            intervalToUpdateSomePages,
            numberOfPagesToUpdateEachInterval
        );

        while (kernel.Processes.Count > 0)
        {
```

```
            kernel.Run();
        }

        Console.WriteLine($"Total requests: {kernel.TotalNumberOfRequests}.");
        Console.WriteLine($"Total page faults: {kernel.NumberOfPagesFault}.");
        Console.WriteLine($"{100 * kernel.NumberOfPagesFault /
kernel.TotalNumberOfRequests}% of requests were page faults.");
    }
}
```

```
namespace lab1
{
    public class VirtualPage
    {
        public bool P { get; set; } // IsPresent
        public bool M { get; set; } // IsModified
        public bool R { get; set; } // IsReferenced
        public uint PPN { get; set; } // PhysicalPageNumber
    }
}
```

```
namespace lab1;

public class PhysicalPage
{
    public uint PPN { get; set; }
    public VirtualPage[]? PageTable { get; set; }
    public int Idx { get; set; }
}
```

```
namespace lab1;
public class WorkingSet
{
    public int SizeOfTable { get; set; }
    public int WorkingSetPercentage { get; set; }
    public int CurrentNumberOfRequests { get; set; } = 0;
    public List<int> IndexesSet { get; set; }

    public WorkingSet(int sizeOfTable, int workingSetPercentage)
    {
        SizeOfTable = sizeOfTable;
        WorkingSetPercentage = workingSetPercentage;
        IndexesSet = new List<int>();
        GenerateNewSet();
    }

    public void GenerateNewSet()
```

```
    {
        var rand = new Random();
        var allIndexes = Enumerable.Range(0, SizeOfTable).ToList();
        var shuffledNumbers = allIndexes.OrderBy(x => rand.Next()).ToList();

        var sizeOfSet = (int)Math.Floor((double)(SizeOfTable * WorkingSetPercentage
/ 100));
        IndexesSet = shuffledNumbers.Take(sizeOfSet).ToList();

        CurrentNumberOfRequests = 0;
    }
}
```

```
namespace lab1;

public class Process
{
    public int Id { get; private set; }
    public VirtualPage[] PageTable { get; private set; }
    public WorkingSet WorkingSet { get; private set; }
    public int RequiredNumberOfRequests { get; private set; }
    public int CurrentNumberOfRequests { get; private set; } = 0;

    public Process(int id, int numberOfVirtualPages, int requiredNumberOfRequests,
int workingSetPercentage)
    {
        Id = id;
        RequiredNumberOfRequests = requiredNumberOfRequests;
        PageTable = new VirtualPage[numberOfVirtualPages];
        WorkingSet = new WorkingSet(numberOfVirtualPages, workingSetPercentage);

        for (int i = 0; i < numberOfVirtualPages; i++)
        {
            PageTable[i] = new VirtualPage
            {
                P = false,
                M = false,
                R = false,
                PPN = 0,
            };
        }

        Console.ForegroundColor = ConsoleColor.Magenta;
        Console.Write("New process");
        Console.ResetColor();
        Console.Write($" with id ");
        Console.ForegroundColor = ConsoleColor.Blue;
        Console.Write($"{id}");
        Console.ResetColor();
```

```
        Console.WriteLine($", {numberOfVirtualPages} virtual pages,
{WorkingSet.IndexesSet.Count} size of working set, {requiredNumberOfRequests}
required number of requests was created.");
    }

    public void IncreaseCurrentRequestsCount(int numberOfRequests)
    {
        CurrentNumberOfRequests += numberOfRequests;
        WorkingSet.CurrentNumberOfRequests += numberOfRequests;
    }

    public bool IsCompleted() => CurrentNumberOfRequests >=
RequiredNumberOfRequests;
}
```

```
namespace lab1;

public class NRUAlgorithm
{
    public readonly List<PhysicalPage>[] ClassifiedPages = new
List<PhysicalPage>[4];
    public NRUAlgorithm()
    {
        for (int i = 0; i < 4; i++)
        {
            ClassifiedPages[i] = new List<PhysicalPage>();
        }
    }

    public void AddPageToAppropriateClass(PhysicalPage page)
    {
        int index = CalculatePageClass(page);
        ClassifiedPages[index].Add(page);
    }

    public PhysicalPage NRUReplacementAlgorithm()
    {
        for (int i = 0; i < ClassifiedPages.Length; i++)
        {
            for (int j = 0; j < ClassifiedPages[i].Count; j++)
            {
                var physicalPage = ClassifiedPages[i][j];

                ClassifiedPages[i].RemoveAt(j);
                j--;

                if (CalculatePageClass(physicalPage) == i)
                {
                    //Console.WriteLine($"Page Fault. Number of page to replace:
{physicalPage.PPN:X8}");
                    return physicalPage;
```

```
            }
            else
            {
                AddPageToAppropriateClass(physicalPage);
            }
        }
    }

    throw new Exception("No pages to replace!");
}

public int CalculatePageClass(PhysicalPage page)
{
    var virtualPage = page.PageTable![page.Idx];

    int index = 0;
    if (virtualPage.R) index += 2;
    if (virtualPage.M) index += 1;

    return index;
}

public void ClearAllReferenceBits()
{
    for (int i = 0; i < ClassifiedPages.Length; i++)
    {
        for (int j = 0; j < ClassifiedPages[i].Count; j++)
        {
            var physicalPage = ClassifiedPages[i][j];
            if (physicalPage.PageTable![physicalPage.Idx].R)
            {
                physicalPage.PageTable![physicalPage.Idx].R = false;
                ClassifiedPages[i].RemoveAt(j);
                j--;
                AddPageToAppropriateClass(physicalPage);
            }
        }
    }

    Console.ForegroundColor = ConsoleColor.DarkRed;
    Console.WriteLine($"All reference bits were cleared.");
    Console.ResetColor();
}

public void UpdateNPages(int n)
{
    int count = 0;
    int i = 0;
    int j = 0;
    int numberOfUpdatedPages = 0;

    while (count < n)
    {
```

```
            if (ClassifiedPages[i].Count == 0)
            {
                i++;
                continue;
            }

            if (j >= ClassifiedPages[i].Count)
            {
                i++;
                j = 0;
            }

            var physicalPage = ClassifiedPages[i][j];

            if (CalculatePageClass(physicalPage) != i)
            {
                ClassifiedPages[i].RemoveAt(j);
                j--;
                AddPageToAppropriateClass(physicalPage);
                numberOfUpdatedPages++;
            }

            j++;
            count++;
        }

        Console.ForegroundColor = ConsoleColor.Green;
        Console.WriteLine($"{numberOfUpdatedPages} pages was updated.");
        Console.ResetColor();
    }
}
```

```
namespace lab1;

public class MMU
{
    public List<PhysicalPage> FreePages;
    // public List<PhysicalPage> BusyPages; // only for Random Replacement
Algolithm
    public NRUAlgorithm NRUAlgorithm; // only for NRU Algolithm

    public MMU(int numberOfPhysicalPages, uint startPageNumber)
    {
        FreePages = new List<PhysicalPage>();
        // BusyPages = new List<PhysicalPage>(); // only for Random Replacement
Algolithm
        NRUAlgorithm = new NRUAlgorithm(); // only for NRU Algolithm

        for (int i = 0; i < numberOfPhysicalPages; i++)
        {
            FreePages.Add(new PhysicalPage { PPN = startPageNumber + (uint)i });
```

```
        }
    }

    public void AccessPage(VirtualPage[] pageTable, int idx, bool isModified,
Kernel kernel)
    {
        PhysicalPage? physicalPage = default;

        if (!pageTable[idx].P)
        {
            physicalPage = kernel.PageFault(pageTable, idx);
        }

        pageTable[idx].R = true;

        if (isModified) pageTable[idx].M = true;

        // only for NRU Algolithm
        if (physicalPage != null)
        {
            NRUAlgorithm.AddPageToAppropriateClass(physicalPage);
        }
    }
}
```

```
namespace lab1;

public class Kernel
{
    public MMU MemoryManager { get; private set; }
    public List<Process> Processes;
    public int NumberOfPagesFault = 0;
    public int TotalNumberOfRequests = 0;
    public int NumberOfRequestsWithThisQuantumOfTime = 0;
    public int NumberOfRequestsFromLastUpdate = 0;
    private int ProcessCount = 0;
    private readonly int StartProcessCount;
    private readonly int MaxProcessCount;
    private readonly int QuantumOfTime;
    private readonly int WorkingSetPercentage;
    private readonly int IntervalToGenerateNewWorkingSet;
    private readonly int IntervalToUpdateSomePages;
    private readonly int NumberOfPagesToUpdateEachInterval;
    private readonly Random Rand = new();

    public Kernel(
        int maxProcessCount,
        int numberOfPhysicalPages,
        uint startPageNumber,
        int startProcessCount,
        int quantumOfTime,
```

```
        int workingSetPercentage,
        int intervalToGenerateNewWorkingSet,
        int intervalToUpdateSomePages,
        int numberOfPagesToUpdateEachInterval
    )
    {
        MemoryManager = new MMU(numberOfPhysicalPages, startPageNumber);
        Processes = new List<Process>();
        StartProcessCount = startProcessCount;
        MaxProcessCount = maxProcessCount;
        QuantumOfTime = quantumOfTime;
        WorkingSetPercentage = workingSetPercentage;
        IntervalToGenerateNewWorkingSet = intervalToGenerateNewWorkingSet;
        IntervalToUpdateSomePages = intervalToUpdateSomePages;
        NumberOfPagesToUpdateEachInterval = numberOfPagesToUpdateEachInterval;

        for (int i = 0; i < startProcessCount; i++)
        {
            AddNewProcess();
        }

        Console.WriteLine($"Kernel with {numberOfPhysicalPages} number of physical
pages, {startProcessCount} start processes, {maxProcessCount} max process count,
{quantumOfTime} quantum of time was created.");
    }

    public void AddNewProcess()
    {
        var id = ProcessCount + 1;
        var numberOfVirtualPages = Rand.Next(30, 60);
        var requiredNumberOfRequests = Rand.Next(300, 600);

        var process = new Process(id, numberOfVirtualPages,
requiredNumberOfRequests, WorkingSetPercentage);
        Processes.Add(process);
        ProcessCount++;
    }

    public PhysicalPage PageFault(VirtualPage[] pageTable, int idx)
    {
        PhysicalPage physicalPage;

        if (MemoryManager.FreePages.Count > 0)
        {
            physicalPage = MemoryManager.FreePages[0];
            MemoryManager.FreePages.RemoveAt(0);
            // MemoryManager.BusyPages.Add(physicalPage); // only for Random
Replacement Algolithm
            // Console.WriteLine("Page Fault.");
        }
        else
        {
```

```
            // physicalPage = RandomReplacementAlgorithm(); // only for Random
Replacement Algolithm
            physicalPage = MemoryManager.NRUAlgorithm.NRUReplacementAlgorithm(); //
only for NRU Algolithm
            physicalPage.PageTable![physicalPage.Idx].P = false;
        }

        physicalPage.PageTable = pageTable;
        physicalPage.Idx = idx;
        pageTable[idx].P = true;
        pageTable[idx].M = false;
        pageTable[idx].PPN = physicalPage.PPN;

        NumberOfPagesFault++;
        return physicalPage;
    }

    // only for Random Replacement Algolithm
    // public PhysicalPage RandomReplacementAlgorithm()
    // {
    //     int index = Rand.Next(MemoryManager.BusyPages.Count);
    //     PhysicalPage pageToReplace = MemoryManager.BusyPages[index];
    //     // Console.WriteLine($"Page Fault. Number of page to replace:
{pageToReplace.PPN:X8}");

    //     return pageToReplace;
    // }

    public void Run()
    {
        for (int i = 0; i < Processes.Count; i++)
        {
            var process = Processes[i];

            if (process.WorkingSet.CurrentNumberOfRequests >=
IntervalToGenerateNewWorkingSet)
            {
                process.WorkingSet.GenerateNewSet();
                Console.ForegroundColor = ConsoleColor.DarkYellow;
                Console.Write("New working set");
                Console.ResetColor();
                Console.Write($" for process ");
                Console.ForegroundColor = ConsoleColor.Blue;
                Console.Write($"{process.Id}");
                Console.ResetColor();
                Console.WriteLine($" was generated: {string.Join(", ",
process.WorkingSet.IndexesSet)}");
            }

            int numberOfRequests = Rand.Next(40, 60);
            int numberOfPagesFaultBefore = NumberOfPagesFault;

            for (int j = 0; j < numberOfRequests; j++)
```

```
            {
                int idx;

                if (Rand.Next(100) < 90)
                {
                    idx =
process.WorkingSet.IndexesSet[Rand.Next(process.WorkingSet.IndexesSet.Count)];
                }
                else
                {
                    idx = Rand.Next(process.PageTable.Length);
                }

                bool isModified = Rand.Next(10) <= 2; // modify/not-modify = 30/70
                MemoryManager.AccessPage(process.PageTable, idx, isModified, this);
            }

            process.IncreaseCurrentRequestsCount(numberOfRequests);
            TotalNumberOfRequests += numberOfRequests;
            NumberOfRequestsWithThisQuantumOfTime += numberOfRequests;
            NumberOfRequestsFromLastUpdate += numberOfRequests;

            int pageFaults = NumberOfPagesFault - numberOfPagesFaultBefore;
            PrintInfo(process, numberOfRequests, pageFaults);

            if (process.IsCompleted())
            {
                Processes.Remove(process);
                i--;
                Console.ForegroundColor = ConsoleColor.Green;
                Console.Write("Finished");
                Console.ResetColor();
                Console.Write($": process with id ");
                Console.ForegroundColor = ConsoleColor.Blue;
                Console.Write($"{process.Id}");
                Console.ResetColor();
                Console.WriteLine(" was completed and removed from queue.");
            }

            if (IsTimeToCreateNewProcess())
            {
                AddNewProcess();
            }

            // only for NRU Algolithm
            if (IsItNewQuantumOfTime())
            {
                MemoryManager.NRUAlgorithm.ClearAllReferenceBits();
                IsTimeToUpdateSomePages();
            }
            else if (IsTimeToUpdateSomePages())
            {
```

```
                MemoryManager.NRUAlgorithm.UpdateNPages(NumberOfPagesToUpdateEachIn
terval);
            }
        }
    }

    private bool IsTimeToUpdateSomePages()
    {
        if (NumberOfRequestsFromLastUpdate >= IntervalToUpdateSomePages)
        {
            NumberOfRequestsFromLastUpdate -= IntervalToUpdateSomePages;
            return true;
        }
        return false;
    }

    private bool IsItNewQuantumOfTime()
    {
        if (NumberOfRequestsWithThisQuantumOfTime >= QuantumOfTime)
        {
            NumberOfRequestsWithThisQuantumOfTime -= QuantumOfTime;
            return true;
        }
        return false;
    }

    private bool IsTimeToCreateNewProcess()
    {
        bool isNewProcessNeeded = ProcessCount < MaxProcessCount;
        bool isNewProcessNeededJustNow = TotalNumberOfRequests - QuantumOfTime *
(ProcessCount - StartProcessCount) >= QuantumOfTime;
        return isNewProcessNeeded && isNewProcessNeededJustNow;
    }

    private void PrintInfo(Process process, int numberOfRequests, int pageFaults)
    {
        Console.WriteLine($"Process with id {process.Id} has made
{numberOfRequests} requests, {pageFaults} of which were page faults.");
        if (MemoryManager.FreePages.Count != 0)
        {
            // Console.WriteLine($"Busy pages: {MemoryManager.BusyPages.Count}.");
// only for Random Replacement Algolithm
            Console.WriteLine($"Busy pages:
{MemoryManager.NRUAlgorithm.ClassifiedPages.Select(list => list.Count).Sum()}.");
// only for NRU Algolithm
            Console.WriteLine($"Free pages: {MemoryManager.FreePages.Count}.");
        }
    }
}
```