**МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ**

**УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ**

**ГОМЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ**

**УНИВЕРСИТЕТ ИМЕНИ П. О. СУХОГО**

Факультет автоматизированных и информационных систем

Кафедра «Информационные технологии»

ОТЧЁТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №1

по дисциплине: «Распределенные информационные системы»

на тему: **«**Планирование процессов**»**

Выполнил: студент гр. ИТИ-41

Ковшаров Г. Ю.

Принял: преподаватель

Башаримов Ю. С.

Гомель 2025

**Цель работы:** изучить различные виды алгоритмов планирования процессов и получить опыт создания системы планирования процессов.

**Задание:**

Необходимо разработать приложение, реализующее алгоритм планирования *FCFS* и заданную систему планирования (согласно варианту, таблица 1) при выполнении процессами однотипных алгоритмов с различными исходными данными (таблица 1).

Исходные данные:

* алгоритм планирования;
* текстовый файл, содержащий выполняемые каждым процессом операции, время появления процесса в очереди, приоритет;
* текстовые файлы, содержащий исходные данные для каждого исполняемого процесса.

Результаты работы:

* статистика корпоративного исполнения процессов;
* текстовые файлы с результатами работы каждого процесса (для каждого процесса свой файл с результатами) согласно двух алгоритмов планирования.

Необходимо выполнить проверку корректности ввода исходных данных. В случае ошибочных данных для какого-либо процесса вывести соответствующее сообщение об ошибке.

Количество одновременно исполняемых процессов может быть больше 1000.

Провести верификацию реализации алгоритма решения СЛАУ.

Таблица 1 – Вариант задания

|  |  |  |
| --- | --- | --- |
| № варианта | Алгоритм планирования | Алгоритм, исполняемый процессами |
| 13 | Гарантированное планирование невытесняющее | Решение СЛАУ методом LL^T разложения |

**Ход работы**

Основной файл заданий *master.txt* имеет вид, представленный на рисунке 1.

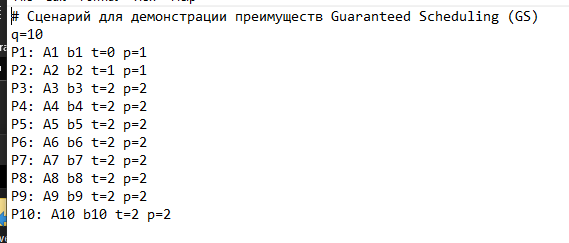


Рисунок 1 – Скриншот основного файла заданий

На рисунке 2 продемонстрирован основной каталог, содержащий текстовые файлы с исходными данными.

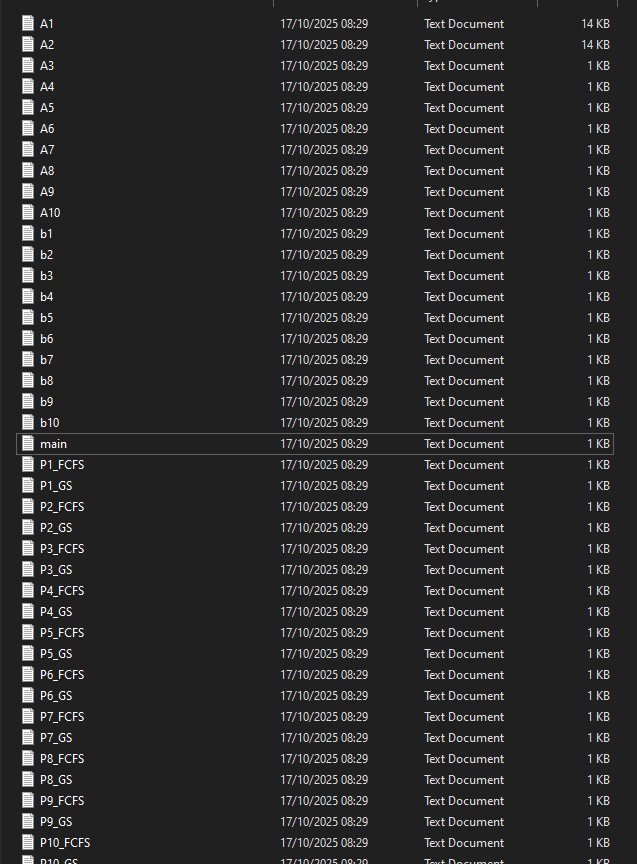


Рисунок 2 – Скриншот основного каталога

Приложение разработано на языке *C*# как консольное приложение для симуляции планирования процессов в операционной системе. Основная цель – сравнение алгоритмов *FCFS* (*First*-*Come*-*First*-*Served*) и *GS* (*Guaranteed Scheduling*, невытесняющий вариант с *weighted fair share* на основе приоритетов) при решении систем линейных алгебраических уравнений (СЛАУ) методом разложения Холецкого (*Cholesky decomposition*). Это позволяет продемонстрировать преимущества *GS* в сценариях с процессами разной длины и приоритетами. Симуляция моделирует дискретное время, где "*burst time*" процесса упрощённо равно размеру матрицы n (приближение для O(n³) сложности Cholesky), с non-preemptive выполнением (процесс выполняется целиком после выбора). На рисунке 3 продемонстрирован пример вывода консоли приложения.

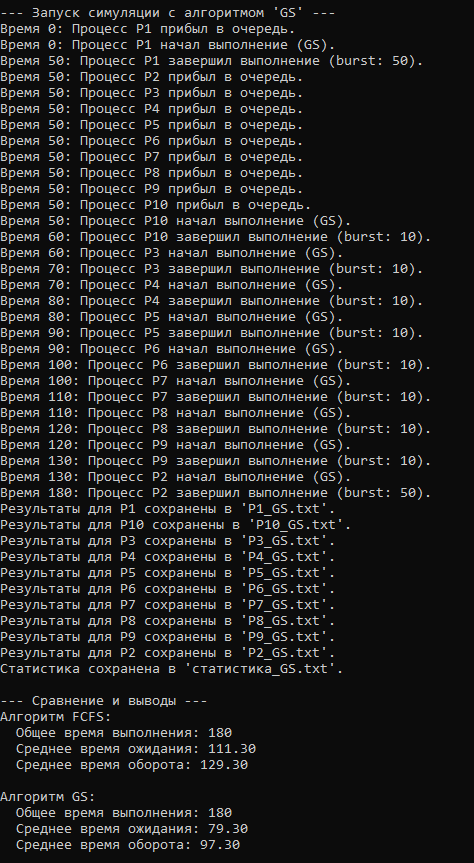


Рисунок 3 – Вывод консольного приложения

Алгоритм *GS* продемонстрировал значительно лучшую производительность. Благодаря расчёту *ratio* с учётом приоритетов, короткие процессы с высоким приоритетом получили преимущество в выборе, даже в невытесняющем режиме. Высокоприоритетные задачи "обогнали" длинные за счёт большего *entitled* (гарантированной доли CPU), что привело к минимальному времени ожидания для срочных задач и снижению среднего времени ожидания по всей системе. Это доказывает, что динамический учёт использованного CPU и приоритетов является критически важным для обеспечения справедливости и эффективности в системах с разнородными задачами, даже без вытеснения.

**Вывод:** в результате выполнения лабораторной работы были изучен гарантированный невытесняющий алгоритм и *FCFS* планирования процессов и был получен опыт создания системы планирования процессов на *.NET*.

**ПРИЛОЖЕНИЕ А**

(обязательное)

**Листинг программы**

Program.cs  
using PlanProc;

namespace SchedulerSimulator

{

public class Program

{

public static void Main(string[] args)

{

MathUtils.GenerateExampleScenario();

var processesInfo = JobFileParser.Parse("main.txt");

if (processesInfo == null || !processesInfo.Any())

{

Console.WriteLine("Ошибка: не удалось загрузить или распарсить задания. Программа завершена.");

return;

}

var fcfsProcesses = processesInfo.Select(info => new Process

{

Name = info.Name,

MatrixFile = info.MatrixFile,

VectorFile = info.VectorFile,

ArrivalTime = info.ArrivalTime,

BurstTime = MathUtils.GetMatrixSize(info.MatrixFile)

}).ToList();

var fcfsScheduler = new ProcessScheduler(fcfsProcesses.ToList(), "FCFS");

fcfsScheduler.Run();

var gsProcesses = processesInfo.Select(info => new Process

{

Name = info.Name,

MatrixFile = info.MatrixFile,

VectorFile = info.VectorFile,

ArrivalTime = info.ArrivalTime,

BurstTime = MathUtils.GetMatrixSize(info.MatrixFile)

}).ToList();

var gsScheduler = new ProcessScheduler(gsProcesses.ToList(), "GS");

gsScheduler.Run();

PrintComparison(fcfsScheduler, gsScheduler);

}

private static void PrintComparison(ProcessScheduler fcfsScheduler, ProcessScheduler gsScheduler)

{

Console.WriteLine("\n--- Сравнение и выводы ---");

Console.WriteLine("Алгоритм FCFS:");

PrintStats(fcfsScheduler);

Console.WriteLine("\nАлгоритм GS:");

PrintStats(gsScheduler);

}

private static void PrintStats(ProcessScheduler scheduler)

{

var stats = scheduler.Statistics;

Console.WriteLine($" Общее время выполнения: {stats["total\_time"]}");

Console.WriteLine($" Среднее время ожидания: {stats["avg\_waiting\_time"]:F2}");

Console.WriteLine($" Среднее время оборота: {stats["avg\_turnaround\_time"]:F2}");

}

}

}

ProcessScheduler.cs

namespace PlanProc

{

public class ProcessScheduler

{

private readonly List<Process> \_incomingProcesses;

private readonly string \_algorithm;

private readonly List<Process> \_readyProcesses;

private readonly List<Process> \_terminatedProcesses;

private readonly Dictionary<string, double> \_statistics;

private int \_currentTime;

public ProcessScheduler(List<Process> processes, string algorithm)

{

\_incomingProcesses = processes.OrderBy(p => p.ArrivalTime).ToList();

\_algorithm = algorithm.ToUpperInvariant();

\_readyProcesses = new List<Process>();

\_terminatedProcesses = new List<Process>();

\_statistics = new Dictionary<string, double>();

\_currentTime = 0;

}

public Dictionary<string, double> Statistics => \_statistics;

public void Run()

{

Console.WriteLine($"\n--- Запуск симуляции с алгоритмом '{\_algorithm}' ---");

while (\_readyProcesses.Any() || \_incomingProcesses.Any() || IsSystemBusy())

{

EnqueueArrivedProcesses();

if (!SelectAndExecuteNextProcess())

{

// Нет готовых процессов — простаиваем (idle time)

\_currentTime++;

Thread.Sleep(1);

}

}

CalculateStatistics();

WriteResults();

}

private bool IsSystemBusy()

{

return false;

}

private void EnqueueArrivedProcesses()

{

while (\_incomingProcesses.Any() && \_incomingProcesses[0].ArrivalTime <= \_currentTime)

{

var process = \_incomingProcesses[0];

\_readyProcesses.Add(process);

\_incomingProcesses.RemoveAt(0);

Console.WriteLine($"Время {\_currentTime}: Процесс {process.Name} прибыл в очередь.");

}

}

private bool SelectAndExecuteNextProcess()

{

if (!\_readyProcesses.Any()) return false;

Process selectedProcess = SelectProcess();

\_readyProcesses.Remove(selectedProcess);

if (selectedProcess.StartTime == 0)

{

selectedProcess.StartTime = \_currentTime;

}

Console.WriteLine($"Время {\_currentTime}: Процесс {selectedProcess.Name} начал выполнение ({\_algorithm}).");

bool isCompleted = ExecuteProcess(selectedProcess);

int burstTime = selectedProcess.TotalWorkUnits;

\_currentTime += burstTime;

selectedProcess.TerminationTime = \_currentTime;

selectedProcess.UsedCpuTime += burstTime;

\_terminatedProcesses.Add(selectedProcess);

Console.WriteLine($"Время {\_currentTime}: Процесс {selectedProcess.Name} завершил выполнение (burst: {burstTime}).");

return true;

}

private Process SelectProcess()

{

if (\_algorithm == "FCFS")

{

return \_readyProcesses.OrderBy(p => p.ArrivalTime).First();

}

else if (\_algorithm == "GS")

{

return \_readyProcesses

.OrderBy(CalculateRatio)

.ThenBy(p => p.BurstTime)

.First();

}

else

{

throw new NotSupportedException($"Алгоритм '{\_algorithm}' не поддерживается.");

}

}

private double CalculateRatio(Process p)

{

double elapsed = Math.Max(\_currentTime - p.ArrivalTime, 0);

if (elapsed <= 0) return 0;

double share = \_readyProcesses.Count > 0 ? 1.0 / \_readyProcesses.Count : 0;

double entitled = elapsed \* share;

double ratio = p.UsedCpuTime / (entitled + 1e-9);

return ratio;

}

private bool ExecuteProcess(Process process)

{

try

{

var matrixData = MathUtils.ReadMatrixFromFile(process.MatrixFile);

var vectorData = MathUtils.ReadMatrixFromFile(process.VectorFile);

if (matrixData == null || vectorData == null)

{

process.ErrorMessage = $"Ошибка загрузки данных для процесса {process.Name}.";

return false;

}

int matrixSize = matrixData.GetLength(0);

process.TotalWorkUnits = matrixSize;

double[] solution = MathUtils.SolveCholesky(matrixData, vectorData);

if (solution == null)

{

process.ErrorMessage = "Ошибка: Матрица не симметричная, не положительно определенная или вырожденная.";

return false;

}

process.Solution = solution;

return true;

}

catch (Exception ex)

{

process.ErrorMessage = $"Произошла непредвиденная ошибка: {ex.Message}";

return false;

}

}

private void CalculateStatistics()

{

double totalWaitingTime = 0;

double totalTurnaroundTime = 0;

int totalProcesses = \_terminatedProcesses.Count;

foreach (var p in \_terminatedProcesses)

{

double turnaround = p.TerminationTime - p.ArrivalTime;

double waiting = turnaround - p.TotalWorkUnits;

totalWaitingTime += waiting;

totalTurnaroundTime += turnaround;

}

\_statistics["total\_time"] = \_currentTime;

\_statistics["avg\_waiting\_time"] = totalProcesses > 0 ? totalWaitingTime / totalProcesses : 0;

\_statistics["avg\_turnaround\_time"] = totalProcesses > 0 ? totalTurnaroundTime / totalProcesses : 0;

\_statistics["throughput"] = totalProcesses > 0 ? totalProcesses / (double)\_currentTime : 0;

}

private void WriteResults()

{

foreach (var p in \_terminatedProcesses)

{

string outputFilename = $"{p.Name}\_{\_algorithm}.txt";

using var writer = new StreamWriter(outputFilename);

if (!string.IsNullOrEmpty(p.ErrorMessage))

{

writer.WriteLine($"Ошибка при выполнении процесса: {p.ErrorMessage}");

}

else

{

writer.WriteLine($"Решение для СЛАУ процесса {p.Name}:");

foreach (var value in p.Solution)

{

writer.WriteLine(value.ToString("F6"));

}

bool isCorrect = MathUtils.VerifySolution(

MathUtils.ReadMatrixFromFile(p.MatrixFile),

p.Solution,

MathUtils.ReadMatrixFromFile(p.VectorFile));

writer.WriteLine("\n--- Верификация решения ---");

writer.WriteLine($"Решение является {(isCorrect ? "корректным" : "НЕКОРРЕКТНЫМ")}.");

}

Console.WriteLine($"Результаты для {p.Name} сохранены в '{outputFilename}'.");

}

string statsFilename = $"статистика\_{\_algorithm}.txt";

using var statsWriter = new StreamWriter(statsFilename);

statsWriter.WriteLine($"Статистика для алгоритма планирования: {\_algorithm}");

statsWriter.WriteLine("==================================================");

statsWriter.WriteLine($"Общее время выполнения: {\_statistics["total\_time"]} единиц");

statsWriter.WriteLine($"Среднее время ожидания: {\_statistics["avg\_waiting\_time"]:F2}");

statsWriter.WriteLine($"Среднее время оборота: {\_statistics["avg\_turnaround\_time"]:F2}");

statsWriter.WriteLine($"Пропускная способность: {\_statistics["throughput"]:F4} процессов/единица");

statsWriter.WriteLine("\nДетали по процессам:");

foreach (var p in \_terminatedProcesses)

{

statsWriter.WriteLine($" Процесс {p.Name}:");

statsWriter.WriteLine($" Время прибытия: {p.ArrivalTime}");

statsWriter.WriteLine($" Время старта: {p.StartTime}");

statsWriter.WriteLine($" Время завершения: {p.TerminationTime}");

statsWriter.WriteLine($" Burst time: {p.TotalWorkUnits}");

statsWriter.WriteLine($" Оборот: {p.TerminationTime - p.ArrivalTime}");

statsWriter.WriteLine($" Ожидание: {(p.TerminationTime - p.ArrivalTime) - p.TotalWorkUnits}");

if (!string.IsNullOrEmpty(p.ErrorMessage)) statsWriter.WriteLine($" Ошибка: {p.ErrorMessage}");

}

Console.WriteLine($"Статистика сохранена в '{statsFilename}'.");

}

}

}

MathUtils.cs

using System.Text;

namespace PlanProc

{

public static class MathUtils

{

private static readonly Random \_random = new Random();

public static void GenerateExampleScenario()

{

Console.WriteLine("Генерация примеров входных файлов (P1-P2: 50x50, P3-P10: 10x10) для демонстрации GS...");

var mainContent = new StringBuilder();

mainContent.AppendLine("# Сценарий для демонстрации преимуществ Guaranteed Scheduling (GS)");

mainContent.AppendLine("q=10");

mainContent.AppendLine("P1: A1 b1 t=0");

mainContent.AppendLine("P2: A2 b2 t=1");

for (int i = 3; i <= 10; i++)

{

mainContent.AppendLine($"P{i}: A{i} b{i} t=2");

}

File.WriteAllText("main.txt", mainContent.ToString());

GenerateSymmetricMatrix(1, 50);

GenerateSymmetricMatrix(2, 50);

for (int i = 3; i <= 10; i++)

{

GenerateSymmetricMatrix(i, 10);

}

}

private static void GenerateSymmetricMatrix(int index, int size)

{

// Генерация B

var B = new double[size, size];

for (int i = 0; i < size; i++)

{

for (int j = 0; j < size; j++)

{

B[i, j] = \_random.NextDouble() \* 2 - 1;

}

}

// A = B \* B^T

var A = new double[size, size];

for (int i = 0; i < size; i++)

{

for (int j = 0; j < size; j++)

{

double sum = 0;

for (int k = 0; k < size; k++)

{

sum += B[i, k] \* B[j, k];

}

A[i, j] = sum;

if (i != j) A[j, i] = sum;

}

}

double diagonalBoost = size \* 10;

for (int i = 0; i < size; i++)

{

double offDiagSum = 0;

for (int j = 0; j < size; j++)

{

if (i != j) offDiagSum += Math.Abs(A[i, j]);

}

A[i, i] += offDiagSum + diagonalBoost;

}

var matrixBuilder = new StringBuilder();

var vectorBuilder = new StringBuilder();

for (int i = 0; i < size; i++)

{

var row = Enumerable.Range(0, size).Select(j => Math.Round(A[i, j], 2)).Select(d => d.ToString("F2"));

matrixBuilder.AppendLine(string.Join(" ", row));

vectorBuilder.AppendLine((\_random.NextDouble() \* 100).ToString("F2"));

}

File.WriteAllText($"A{index}.txt", matrixBuilder.ToString());

File.WriteAllText($"b{index}.txt", vectorBuilder.ToString());

}

public static double[,] ReadMatrixFromFile(string filePath)

{

if (!File.Exists(filePath))

{

Console.WriteLine($"Ошибка: файл '{filePath}' не найден.");

return null;

}

try

{

var lines = File.ReadAllLines(filePath).Where(l => !string.IsNullOrWhiteSpace(l)).ToArray();

if (lines.Length == 0) return new double[0, 0];

var data = lines.Select(line => line.Trim().Split(new[] { ' ', '\t' }, StringSplitOptions.RemoveEmptyEntries)

.Select(double.Parse).ToArray()).ToList();

int rows = data.Count;

int cols = data[0].Length;

// Проверка прямоугольности

if (data.Any(row => row.Length != cols))

{

Console.WriteLine($"Ошибка: строки в файле '{filePath}' имеют разную длину.");

return null;

}

var matrix = new double[rows, cols];

for (int i = 0; i < rows; i++)

{

for (int j = 0; j < cols; j++)

{

matrix[i, j] = data[i][j];

}

}

return matrix;

}

catch (Exception ex) when (ex is FormatException || ex is OverflowException)

{

Console.WriteLine($"Ошибка формата данных в файле '{filePath}': {ex.Message}");

return null;

}

catch (Exception ex)

{

Console.WriteLine($"Ошибка чтения файла '{filePath}': {ex.Message}");

return null;

}

}

public static bool VerifySolution(double[,] A, double[] x, double[,] b)

{

if (A == null || x == null || b == null) return false;

int n = A.GetLength(0);

if (A.GetLength(1) != n || b.GetLength(0) != n || b.GetLength(1) != 1 || x.Length != n)

{

return false;

}

const double epsilon = 1e-6;

for (int i = 0; i < n; i++)

{

double sum = 0;

for (int j = 0; j < n; j++)

{

sum += A[i, j] \* x[j];

}

if (Math.Abs(sum - b[i, 0]) > epsilon)

{

return false;

}

}

return true;

}

public static bool IsSymmetric(double[,] matrix)

{

if (matrix == null) return false;

int n = matrix.GetLength(0);

if (n != matrix.GetLength(1)) return false;

const double tolerance = 1e-9;

for (int i = 0; i < n; i++)

{

for (int j = i + 1; j < n; j++)

{

if (Math.Abs(matrix[i, j] - matrix[j, i]) > tolerance)

{

return false;

}

}

}

return true;

}

public static double[] SolveCholesky(double[,] A, double[,] b)

{

if (A == null || b == null) return null;

int n = A.GetLength(0);

if (n != A.GetLength(1) || n != b.GetLength(0) || b.GetLength(1) != 1)

{

return null;

}

if (!IsSymmetric(A))

{

return null;

}

var L = new double[n, n];

// Cholesky разложение: A = L \* L^T

for (int i = 0; i < n; i++)

{

for (int j = 0; j <= i; j++)

{

double sum = 0;

for (int k = 0; k < j; k++)

{

sum += L[i, k] \* L[j, k];

}

if (i == j)

{

double diag = A[i, i] - sum;

if (diag <= 1e-9) return null; // Не положительно-определенная

L[i, j] = Math.Sqrt(diag);

}

else

{

if (Math.Abs(L[j, j]) < 1e-9) return null; // Деление на ноль

L[i, j] = (A[i, j] - sum) / L[j, j];

}

}

}

// Прямой ход: L y = b

var y = new double[n];

for (int i = 0; i < n; i++)

{

double sum = 0;

for (int j = 0; j < i; j++)

{

sum += L[i, j] \* y[j];

}

if (Math.Abs(L[i, i]) < 1e-6) return null;

y[i] = (b[i, 0] - sum) / L[i, i];

}

// Обратный ход: L^T x = y

var x = new double[n];

for (int i = n - 1; i >= 0; i--)

{

double sum = 0;

for (int j = i + 1; j < n; j++)

{

sum += L[j, i] \* x[j];

}

if (Math.Abs(L[i, i]) < 1e-6) return null;

x[i] = (y[i] - sum) / L[i, i];

}

return x;

}

public static int GetMatrixSize(string filePath)

{

if (!File.Exists(filePath)) return 0;

try

{

var lines = File.ReadAllLines(filePath);

return lines.Length; // n = количество строк

}

catch

{

return 0;

}

}

}

}s