МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»

ФАКУЛЬТЕТ ІНФОРМАТИКИ ТА ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ

Кафедра інформатики та програмної інженерії

**Звіт**

З лабораторної роботи № 3 з дисципліни

«Моделювання систем»

«**Побудова імітаційної моделі системи з використанням**

**формалізму моделі масового обслуговування**»

| **Виконав(ла)** | *ІП-13 Бабіч Денис* |  |  |
| --- | --- | --- | --- |
|  | (шифр, прізвище, ім'я, по батькові) |  |  |

| **Перевірив(ла)** | *Дифучин А. Ю.* |  |  |
| --- | --- | --- | --- |
|  | (посада, прізвище, ім'я, по батькові) |  |  |

Київ 2024

# ОСНОВНА ЧАСТИНА

**Мета роботи**: Побудувати імітаційні моделі системи з використанням формалізму моделі масового обслуговування.

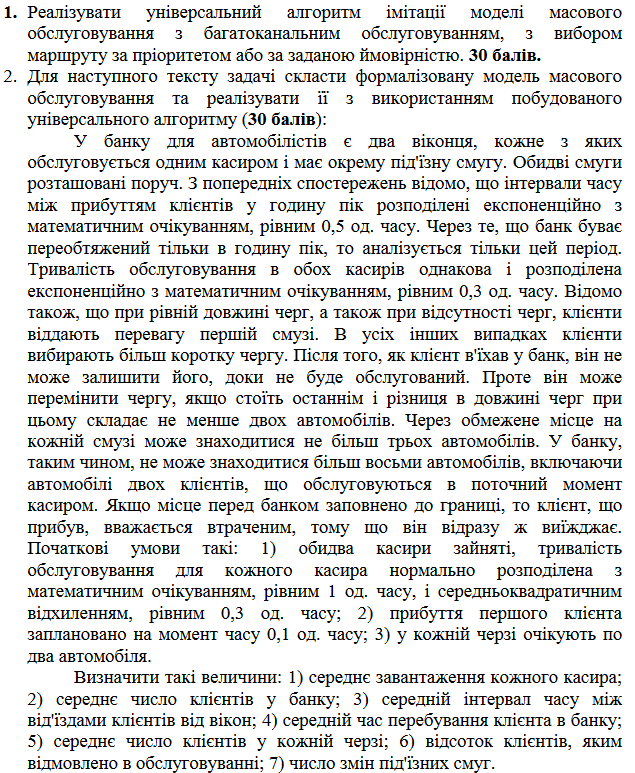
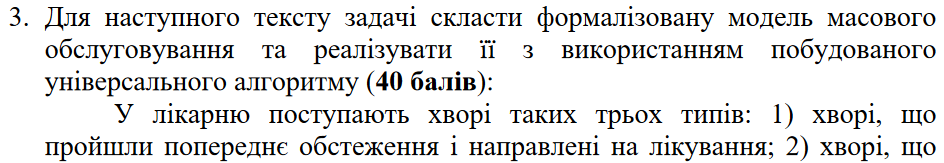


Рисунок 1.1 – Завдання лабораторного практикуму



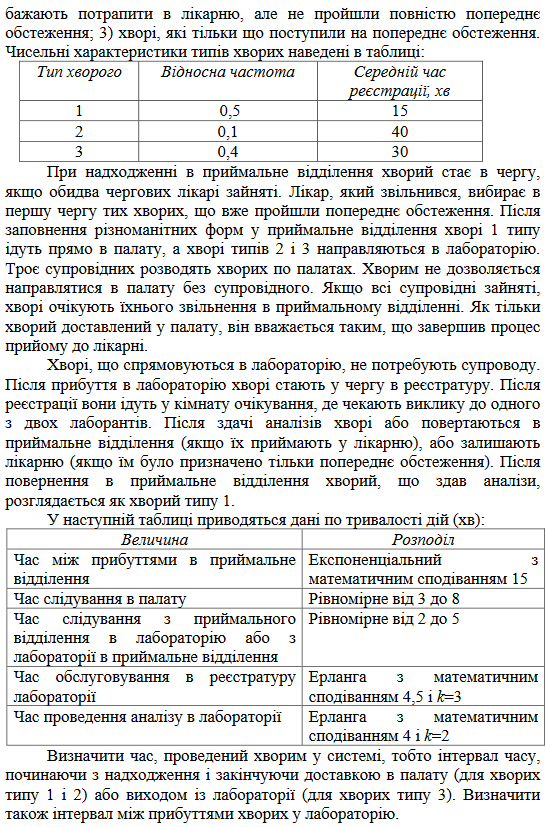


Рисунок 1.3 – Завдання лабораторного практикуму

# 

Рисунок 1.4 – Формалізм моделі першого завдання

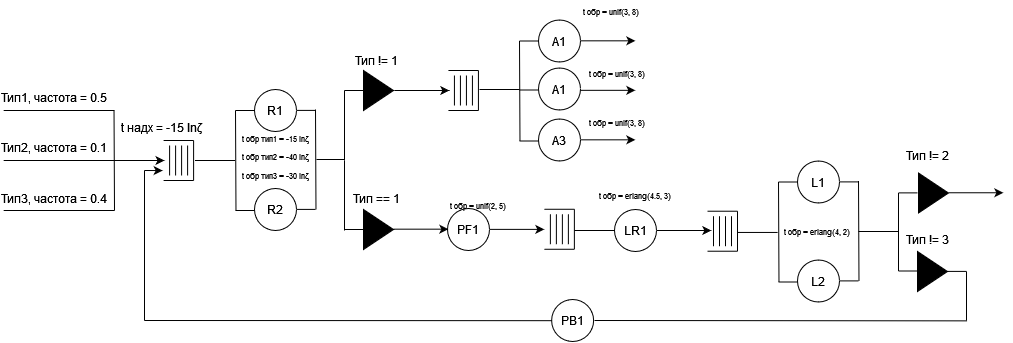


Рисунок 1.5 – Формалізм моделі другого завдання

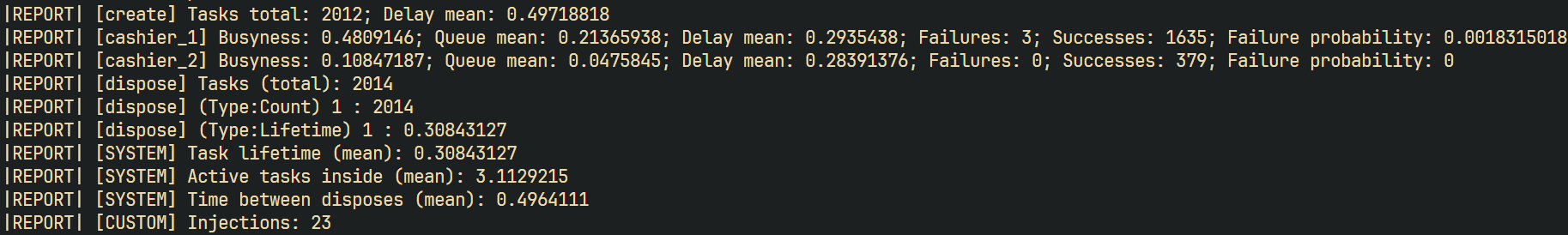


Рисунок 1.6 – Результати моделювання першої моделі

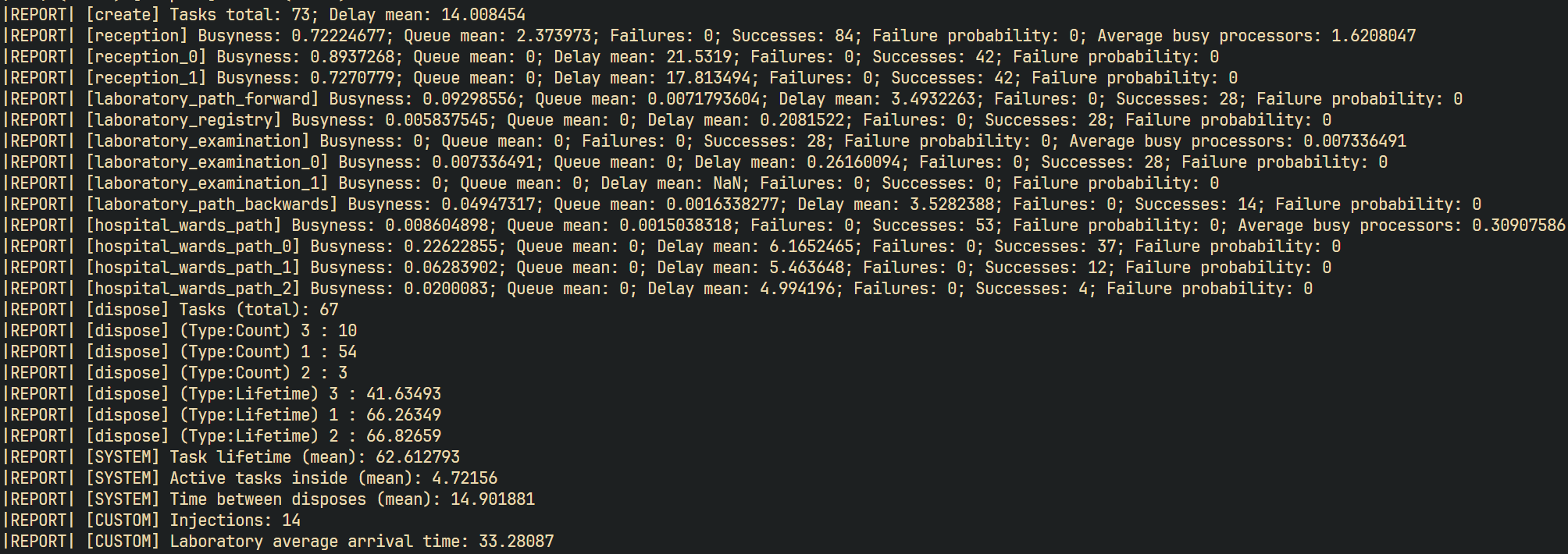


Рисунок 1.7 – Результати моделювання другої моделі

# ВИСНОВКИ

В ході виконання роботи було розроблено універсальну модель масового обслуговування, що враховує багатоканальне обслуговування з можливістю вибору маршруту за пріоритетом або заданою ймовірністю. Це дозволило створити гнучкий алгоритм для моделювання систем з обмеженнями на кількість клієнтів та правилами перемикання між чергами. Така модель відповідає завданню на прикладі банку, де клієнти вибирають смуги обслуговування на основі чергових пріоритетів, дотримуючись обмежень на кількість автомобілів у системі.

Реалізація моделі була продемонстрована на прикладі двох завдань: роботи банку для автомобілістів з двома каналами обслуговування та лікарні з кількома типами пацієнтів і різними етапами обслуговування. Модель ефективно враховує типи клієнтів, відмінності у часі прибуття, перебування та обробки заявок, дозволяючи виявити важливі показники, такі як середнє завантаження каналів, середню кількість клієнтів у чергах, частоту відмов у обслуговуванні та інші ключові параметри системи.

Отримані результати підтвердили коректність алгоритму та відповідність вимогам завдання. Модель продемонструвала надійність у розрахунку показників, необхідних для оцінки ефективності обслуговування, зокрема середнього часу перебування клієнтів, завантаження каналів і частоти перенаправлень. Такий підхід відкриває можливість подальшої адаптації моделі для складніших сценаріїв і розширення її застосування у моделюванні систем обслуговування різної складності.

# ДОДАТОК ПРОГРАМНИЙ КОД

Programs.cs

using System;

using System.Collections.Generic;

using LabWork3.Framework.Core.Controllers;

using LabWork3.Framework.Components.Tasks.Common;

using LabWork3.Framework.Components.Tasks.Concrete;

using LabWork3.Framework.Components.Queues.Concrete;

using LabWork3.Framework.Components.Modules.Common;

using LabWork3.Framework.Components.Modules.Concrete;

using LabWork3.Framework.Components.Workers.Common;

using LabWork3.Framework.Components.Workers.Concrete;

using LabWork3.Framework.Components.Schemes.Concrete;

using LabWork2.Framework.Components.Modules.Concrete;

using LabWork3.Framework.Components.Tasks.Utilities.Factories.Concrete;

namespace LabWork3.Application;

file sealed class Program

{

private static void Main()

{

// Program.RunCarBankModel();

Program.RunHospitalModel();

}

private static void RunCarBankModel()

{

const int MAX\_QUEUE\_LENGTH = 3;

int injectionsCount = 0;

CreateModule create;

DisposeModule dispose;

ProcessorModule cashier1;

ProcessorModule cashier2;

dispose = new DisposeModule("dispose");

DefaultQueue cashier1Queue = new DefaultQueue(MAX\_QUEUE\_LENGTH);

cashier1 = new ProcessorModule("cashier\_1", new SingleTransitionScheme(dispose), new MockExponentialWorker(0.3f), cashier1Queue);

DefaultQueue cashier2Queue = new DefaultQueue(MAX\_QUEUE\_LENGTH);

cashier2 = new ProcessorModule("cashier\_2", new SingleTransitionScheme(dispose), new MockExponentialWorker(0.3f), cashier2Queue);

QueuePriorityScheme createScheme = new QueuePriorityScheme(dispose, InjectCustomLogic);

createScheme.Attach(cashier1);

createScheme.Attach(cashier2);

create = new CreateModule("create", createScheme, new MockExponentialWorker(0.5f), new CarTaskFactory(), 0.1f);

cashier1Queue.AddLast(new CarTask(0.0f));

cashier1Queue.AddLast(new CarTask(0.0f));

cashier1.AcceptInitialTask(new CarTask(0.0f), new MockNormalWorker(1, 0.3f).DelayPayload);

cashier2Queue.AddLast(new CarTask(0.0f));

cashier2Queue.AddLast(new CarTask(0.0f));

cashier2.AcceptInitialTask(new CarTask(0.0f), new MockNormalWorker(1, 0.3f).DelayPayload);

cashier1Queue.TaskAdded += OnTaskAdded;

cashier1Queue.TaskRemoved += OnTaskRemoved;

cashier2Queue.TaskAdded += OnTaskAdded;

cashier2Queue.TaskRemoved += OnTaskRemoved;

new SimulationModelController(new Module[] { create, cashier1, cashier2, dispose }).RunSimulation(1000.0f);

Console.WriteLine($"|REPORT| [CUSTOM] Injections: {injectionsCount}");

cashier1Queue.TaskAdded -= OnTaskAdded;

cashier1Queue.TaskRemoved -= OnTaskRemoved;

cashier2Queue.TaskAdded -= OnTaskAdded;

cashier2Queue.TaskRemoved -= OnTaskRemoved;

ProcessorModule? InjectCustomLogic()

{

if (cashier1.Queue.Count == 0)

return cashier1;

if ((cashier1.Queue.Count != MAX\_QUEUE\_LENGTH) && (cashier1.Queue.Count == cashier2.Queue.Count))

return cashier1;

return null;

}

void OnTaskAdded(object? sender, EventArgs eventArgs)

{

BalanceQueues(cashier1, cashier2);

}

void OnTaskRemoved(object? sender, EventArgs eventArgs)

{

BalanceQueues(cashier1, cashier2);

}

void BalanceQueues(ProcessorModule source, ProcessorModule target)

{

const int TARGET\_DELTA\_QUEUES\_LENGTH = 2;

int deltaQueueLength = source.Queue.Count - target.Queue.Count;

if (Math.Abs(deltaQueueLength) >= TARGET\_DELTA\_QUEUES\_LENGTH)

{

if (deltaQueueLength > 0)

{

Task task = source.Queue.RemoveLast();

target.Queue.AddLast(task);

}

else if (deltaQueueLength < 0)

{

Task task = target.Queue.RemoveLast();

source.Queue.AddLast(task);

}

++injectionsCount;

}

}

}

private static void RunHospitalModel()

{

const int PATIENT\_TYPE\_1 = 1;

const int PATIENT\_TYPE\_2 = 2;

const int PATIENT\_TYPE\_3 = 3;

int injectionsCount = 0;

CreateModule create;

DisposeModule dispose;

CustomMultiProcessorModule reception;

ProcessorModule laboratoryPathForward;

ProcessorModule laboratoryPathBackwards;

ProcessorModule laboratoryRegistry;

MultiProcessorModule hospitalWardsPath;

MultiProcessorModule laboratoryExamination;

dispose = new DisposeModule("dispose");

TypeScheme receptionScheme = new TypeScheme(dispose);

Dictionary<int, IMockWorker> receptionWorkers = new Dictionary<int, IMockWorker>();

receptionWorkers[PATIENT\_TYPE\_1] = new MockExponentialWorker(15);

receptionWorkers[PATIENT\_TYPE\_2] = new MockExponentialWorker(40);

receptionWorkers[PATIENT\_TYPE\_3] = new MockExponentialWorker(30);

reception = new CustomMultiProcessorModule("reception", receptionScheme, receptionWorkers, new DefaultQueue(Int32.MaxValue), 2);

hospitalWardsPath = new MultiProcessorModule("hospital\_wards\_path", new SingleTransitionScheme(dispose), new MockUniformWorker(3, 8), new DefaultQueue(Int32.MaxValue), 3);

ProbabilityScheme laboratoryExaminationScheme = new ProbabilityScheme(dispose);

laboratoryExamination = new MultiProcessorModule("laboratory\_examination", laboratoryExaminationScheme, new MockErlangWorker(4.0f, 2), new DefaultQueue(Int32.MaxValue), 2);

laboratoryRegistry = new ProcessorModule("laboratory\_registry", new SingleTransitionScheme(laboratoryExamination, dispose), new MockErlangWorker(4.5f, 3), new DefaultQueue(Int32.MaxValue));

laboratoryPathForward = new ProcessorModule("laboratory\_path\_forward", new SingleTransitionScheme(laboratoryRegistry, dispose), new MockUniformWorker(2, 5), new DefaultQueue(Int32.MaxValue));

laboratoryPathBackwards = new ProcessorModule("laboratory\_path\_backwards", new SingleTransitionScheme(reception, dispose), new MockUniformWorker(2, 5), new DefaultQueue(Int32.MaxValue));

laboratoryExaminationScheme.Attach(dispose, 0.5f);

laboratoryExaminationScheme.Attach(laboratoryPathBackwards, 0.5f, InjectCustomLogic);

receptionScheme.Attach(hospitalWardsPath, PATIENT\_TYPE\_1);

receptionScheme.Attach(laboratoryPathForward, PATIENT\_TYPE\_2);

receptionScheme.Attach(laboratoryPathForward, PATIENT\_TYPE\_3);

create = new CreateModule("create", new SingleTransitionScheme(reception, dispose), new MockExponentialWorker(15), new PatientTaskFactory());

new SimulationModelController(new Module[] { create, reception, laboratoryPathForward, laboratoryRegistry, laboratoryExamination, laboratoryPathBackwards, hospitalWardsPath, dispose }).RunSimulation(1000.0f);

Console.WriteLine($"|REPORT| [CUSTOM] Injections: {injectionsCount}");

float totalVisitorsCount = laboratoryExamination.Queue.Count + laboratoryExamination.SuccessesCount + laboratoryExamination.BusySubProcessors.Count;

Console.WriteLine($"|REPORT| [CUSTOM] Laboratory average arrival time: {dispose.TimeCurrent / totalVisitorsCount}");

void InjectCustomLogic(Task task)

{

++injectionsCount;

task.CurrentType = PATIENT\_TYPE\_1;

}

}

}

CreateModule.cs

using System;

using LabWork3.Framework.Components.Tasks.Common;

using LabWork3.Framework.Components.Schemes.Common;

using LabWork3.Framework.Components.Workers.Common;

using LabWork3.Framework.Components.Modules.Common;

using LabWork3.Framework.Components.Tasks.Utilities.Factories.Common;

namespace LabWork3.Framework.Components.Modules.Concrete;

internal sealed class CreateModule : Module

{

private readonly IScheme scheme;

private readonly IMockWorker mockWorker;

private readonly TaskFactory taskFactory;

private int createdTasksCount;

private float totalDelayPayloads;

internal CreateModule(string identifier, IScheme scheme, IMockWorker mockWorker, TaskFactory taskFactory, float initialTime = 0.0f) : base(identifier)

{

if (scheme == null)

throw new ArgumentNullException($"{nameof(scheme)} cannot be null.");

if (mockWorker == null)

throw new ArgumentNullException($"{nameof(mockWorker)} cannot be null.");

if (taskFactory == null)

throw new ArgumentNullException($"{nameof(taskFactory)} cannot be null.");

this.scheme = scheme;

this.mockWorker = mockWorker;

this.taskFactory = taskFactory;

this.MoveTimeline((initialTime != 0.0f ? initialTime : mockWorker.DelayPayload));

}

internal int CreatedTasksCount => this.createdTasksCount;

internal override sealed float TimeCurrent { get; set; }

private protected override sealed void MoveTimeline(float deltaTime)

{

this.totalDelayPayloads += deltaTime;

base.TimeNext = this.TimeCurrent + deltaTime;

}

internal override sealed void AcceptTask(Task task, IMockWorker? mockWorker)

{

throw new InvalidOperationException($"{base.Identifier} ({this.GetType()}) is not able to accept tasks.");

}

internal override sealed void CompleteTask()

{

++this.createdTasksCount;

Task newTask = this.taskFactory.CreateTask(this.TimeCurrent);

Module? nextModule = this.scheme.GetNextModule(newTask);

nextModule?.AcceptTask(newTask, null);

this.MoveTimeline(this.mockWorker.DelayPayload);

Console.WriteLine($"|LOG| (TRACE) [{base.Identifier}] sends task to the [{nextModule?.Identifier}]");

}

public override sealed void PrintIntermediateStatistics()

{

Console.Write($"|LOG| (STATS) [{base.Identifier}] ");

Console.WriteLine($"Tasks: {this.createdTasksCount}; Time: {base.TimeNext}.");

}

public override sealed void PrintFinalStatistics()

{

Console.Write($"|REPORT| [{base.Identifier}] ");

Console.WriteLine($"Tasks total: {this.createdTasksCount}; Delay mean: {this.totalDelayPayloads / this.createdTasksCount}");

}

}

Processor.cs

using System;

using LabWork3.Framework.Components.Tasks.Common;

using LabWork3.Framework.Components.Queues.Common;

using LabWork3.Framework.Components.Schemes.Common;

using LabWork3.Framework.Components.Modules.Common;

using LabWork3.Framework.Components.Workers.Common;

namespace LabWork3.Framework.Components.Modules.Concrete;

internal sealed class ProcessorModule : Module

{

private readonly IScheme scheme;

private float timeCurrent;

private Task? currentTask;

private IMockWorker? mockWorker;

private float totalTimeBusy;

private float totalDelayPayloads;

private float totalQueueLengthSum;

internal ProcessorModule(string identifier, IScheme scheme, IMockWorker? mockWorker, IQueue queue) : base(identifier)

{

if (queue == null)

throw new ArgumentNullException($"{nameof(queue)} cannot be null.");

if (scheme == null)

throw new ArgumentNullException($"{nameof(scheme)} cannot be null.");

this.Queue = queue;

this.scheme = scheme;

this.mockWorker = mockWorker;

}

internal bool IsBusy { get; private set; }

internal IQueue Queue { get; private init; }

internal int FailuresCount { get; private set; }

internal int SuccessesCount { get; private set; }

internal int CurrentTasksCount => this.Queue.Count + (this.IsBusy ? 1 : 0);

internal override sealed float TimeCurrent

{

get => this.timeCurrent;

set

{

float deltaTime = value - this.timeCurrent;

this.totalTimeBusy += (this.IsBusy ? deltaTime : 0.0f);

this.totalQueueLengthSum += deltaTime \* this.Queue.Count;

this.timeCurrent = value;

}

}

internal override sealed void AcceptTask(Task task, IMockWorker? customMockWorker)

{

if (this.IsBusy)

{

if (!this.Queue.IsFull)

this.Queue.AddLast(task);

else

++this.FailuresCount;

}

else

{

this.IsBusy = true;

this.currentTask = task;

this.mockWorker = customMockWorker != null ? customMockWorker : this.mockWorker;

this.MoveTimeline(this.mockWorker!.DelayPayload);

}

}

internal void AcceptInitialTask(Task task, float delayPayload)

{

this.IsBusy = true;

this.currentTask = task;

base.TimeNext = delayPayload;

}

internal override sealed void CompleteTask()

{

++this.SuccessesCount;

if (this.Queue.IsEmpty)

{

this.IsBusy = false;

this.TimeNext = Single.MaxValue;

}

else

{

this.currentTask = this.Queue.RemoveFirst();

this.MoveTimeline(this.mockWorker!.DelayPayload);

}

Module? nextModule = this.scheme.GetNextModule(this.currentTask!);

nextModule?.AcceptTask(this.currentTask!, null);

Console.WriteLine($"|LOG| (TRACE) [{base.Identifier}] sends task to the [{nextModule?.Identifier}]");

}

private protected override sealed void MoveTimeline(float deltaTime)

{

this.totalDelayPayloads += deltaTime;

this.TimeNext = this.TimeCurrent + deltaTime;

}

public override sealed void PrintIntermediateStatistics()

{

Console.Write($"|LOG| (STATS) [{base.Identifier}] ");

Console.WriteLine($"Busy?: {this.IsBusy}; Queue: {this.Queue.Count}; Successes: {this.SuccessesCount}; Failures: {this.FailuresCount}");

}

public override sealed void PrintFinalStatistics()

{

float busyness = this.totalTimeBusy / this.TimeCurrent;

float queueLengthMean = this.totalQueueLengthSum / this.TimeCurrent;

float delayPayloadMean = this.totalDelayPayloads / this.SuccessesCount;

float failureProbability = (this.SuccessesCount == 0 ? 0 : (float)this.FailuresCount / (this.FailuresCount + this.SuccessesCount));

Console.Write($"|REPORT| [{base.Identifier}] ");

Console.Write($"Busyness: {busyness}; ");

Console.Write($"Queue mean: {queueLengthMean}; ");

Console.Write($"Delay mean: {delayPayloadMean}; ");

Console.Write($"Failures: {this.FailuresCount}; ");

Console.Write($"Successes: {this.SuccessesCount}; ");

Console.WriteLine($"Failure probability: {failureProbability}");

}

}

MultiProcessorModule.cs

using System;

using System.Linq;

using System.Collections.Generic;

using LabWork3.Framework.Components.Tasks.Common;

using LabWork3.Framework.Components.Queues.Common;

using LabWork3.Framework.Components.Queues.Concrete;

using LabWork3.Framework.Components.Workers.Common;

using LabWork3.Framework.Components.Schemes.Common;

using LabWork3.Framework.Components.Modules.Common;

using LabWork3.Framework.Components.Modules.Concrete;

using LabWork3.Framework.Components.Schemes.Concrete;

namespace LabWork2.Framework.Components.Modules.Concrete;

internal sealed class MultiProcessorModule : Module

{

private readonly IScheme distribution;

private readonly IMockWorker mockWorker;

private readonly IList<ProcessorModule> subProcessors;

private float timeCurrent;

private float totalTimeBusy;

private float totalQueueLengthSum;

private float totalSubProcessorsTimeBusy;

internal MultiProcessorModule(string identifier, IScheme scheme, IMockWorker mockWorker, IQueue queue, int subProcessorsCount) : base(identifier)

{

if (scheme == null)

throw new ArgumentNullException($"{nameof(scheme)} cannot be null.");

if (mockWorker == null)

throw new ArgumentNullException($"{nameof(mockWorker)} cannot be null.");

if (queue == null)

throw new ArgumentNullException($"{nameof(queue)} cannot be null.");

if (subProcessorsCount <= 0)

throw new ArgumentException($"{nameof(subProcessorsCount)} cannot be less or equals 0.");

this.Queue = queue;

this.mockWorker = mockWorker;

this.subProcessors = new ProcessorModule[subProcessorsCount];

this.distribution = new PayloadDistributionScheme(this.subProcessors, scheme.Fallback);

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

this.subProcessors[i] = new ProcessorModule($"{identifier}\_{i}", scheme, mockWorker, new DefaultQueue(0));

}

internal IQueue Queue { get; private init; }

internal int FailuresCount { get; private set; }

internal int SuccessesCount { get; private set; }

internal bool IsPartiallyBusy => this.subProcessors.Any(subProcessorModule => subProcessorModule.IsBusy);

internal bool IsCompletelyBusy => this.subProcessors.All(subProcessorModule => subProcessorModule.IsBusy);

internal List<ProcessorModule> BusySubProcessors => this.subProcessors.Where(processor => processor.TimeNext == this.TimeNext).ToList();

internal override sealed float TimeCurrent

{

get => this.timeCurrent;

set

{

float deltaTime = value - this.timeCurrent;

this.totalSubProcessorsTimeBusy += this.subProcessors.Count(processor => processor.IsBusy) \* deltaTime;

this.totalTimeBusy += (this.IsCompletelyBusy ? deltaTime : 0.0f);

this.totalQueueLengthSum += deltaTime \* this.Queue.Count;

this.timeCurrent = value;

foreach (ProcessorModule processor in this.subProcessors)

processor.TimeCurrent = this.timeCurrent;

}

}

internal override sealed void AcceptTask(Task task, IMockWorker? mockWorker)

{

if (this.IsCompletelyBusy)

{

if (!this.Queue.IsFull)

this.Queue.AddLast(task);

else

++this.FailuresCount;

}

else

{

Module? nextModule = this.distribution.GetNextModule(task);

nextModule?.AcceptTask(task, null);

Console.WriteLine($"|LOG| (TRACE) [{base.Identifier}] sends task to the [{nextModule?.Identifier}]");

this.MoveTimeline(this.mockWorker.DelayPayload);

}

}

internal override sealed void CompleteTask()

{

++this.SuccessesCount;

List<ProcessorModule> BusySubProcessors = this.BusySubProcessors;

foreach (ProcessorModule processor in BusySubProcessors)

{

processor.CompleteTask();

if (!this.Queue.IsEmpty)

{

Task newTask = this.Queue.RemoveFirst();

processor.AcceptTask(newTask, null);

Console.WriteLine($"|LOG| (TRACE) [{base.Identifier}] sends task to the [{processor.Identifier}]");

}

}

this.MoveTimeline(this.mockWorker.DelayPayload);

}

private protected override sealed void MoveTimeline(float deltaTime)

{

this.TimeNext = this.subProcessors.Min(processor => processor.TimeNext);

}

public override sealed void PrintIntermediateStatistics()

{

Console.Write($"|LOG| (STATS) [{base.Identifier}] ");

Console.WriteLine($"Busy sub-processors: {this.BusySubProcessors.Count}; Queue: {this.Queue.Count}; Successes: {this.SuccessesCount}; Failures: {this.FailuresCount}");

}

public override sealed void PrintFinalStatistics()

{

float busyness = this.totalTimeBusy / this.TimeCurrent;

float queueLengthMean = this.totalQueueLengthSum / this.TimeCurrent;

float failureProbability = (this.SuccessesCount == 0 ? 0 : (float)this.FailuresCount / (this.FailuresCount + this.SuccessesCount));

Console.Write($"|REPORT| [{base.Identifier}] ");

Console.Write($"Busyness: {busyness}; ");

Console.Write($"Queue mean: {queueLengthMean}; ");

Console.Write($"Failures: {this.FailuresCount}; ");

Console.Write($"Successes: {this.SuccessesCount}; ");

Console.Write($"Failure probability: {failureProbability}; ");

Console.WriteLine($"Average busy processors: {this.totalSubProcessorsTimeBusy / this.TimeCurrent}");

foreach (ProcessorModule processorModule in this.subProcessors)

processorModule.PrintFinalStatistics();

}

}

CustomMultiProcessorModule

using System;

using System.Linq;

using System.Collections.Generic;

using LabWork3.Framework.Components.Tasks.Common;

using LabWork3.Framework.Components.Queues.Common;

using LabWork3.Framework.Components.Queues.Concrete;

using LabWork3.Framework.Components.Workers.Common;

using LabWork3.Framework.Components.Schemes.Common;

using LabWork3.Framework.Components.Modules.Common;

using LabWork3.Framework.Components.Modules.Concrete;

using LabWork3.Framework.Components.Schemes.Concrete;

namespace LabWork2.Framework.Components.Modules.Concrete;

internal sealed class CustomMultiProcessorModule : Module

{

private readonly IScheme distribution;

private readonly IList<ProcessorModule> subProcessors;

private readonly IDictionary<int, IMockWorker> mockWorkers;

private float timeCurrent;

private float totalTimeBusy;

private float totalQueueLengthSum;

private float totalSubProcessorsTimeBusy;

internal CustomMultiProcessorModule(string identifier, IScheme scheme, IDictionary<int, IMockWorker> mockWorkers, IQueue queue, int subProcessorsCount) : base(identifier)

{

if (scheme == null)

throw new ArgumentNullException($"{nameof(scheme)} cannot be null.");

if (mockWorkers == null)

throw new ArgumentNullException($"{nameof(mockWorkers)} cannot be null.");

if (queue == null)

throw new ArgumentNullException($"{nameof(queue)} cannot be null.");

if (subProcessorsCount <= 0)

throw new ArgumentException($"{nameof(subProcessorsCount)} cannot be less or equals 0.");

this.Queue = queue;

this.mockWorkers = mockWorkers;

this.subProcessors = new ProcessorModule[subProcessorsCount];

this.distribution = new PayloadDistributionScheme(this.subProcessors, scheme.Fallback);

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

this.subProcessors[i] = new ProcessorModule($"{identifier}\_{i}", scheme, mockWorkers.Values.FirstOrDefault(), new DefaultQueue(0));

}

internal IQueue Queue { get; private init; }

internal int FailuresCount { get; private set; }

internal int SuccessesCount { get; private set; }

internal bool IsPartiallyBusy => this.subProcessors.Any(subProcessorModule => subProcessorModule.IsBusy);

internal bool IsCompletelyBusy => this.subProcessors.All(subProcessorModule => subProcessorModule.IsBusy);

internal List<ProcessorModule> BusySubProcessors => this.subProcessors.Where(processor => processor.TimeNext == this.TimeNext).ToList();

internal override sealed float TimeCurrent

{

get => this.timeCurrent;

set

{

float deltaTime = value - this.timeCurrent;

this.totalSubProcessorsTimeBusy += this.subProcessors.Count(processor => processor.IsBusy) \* deltaTime;

this.totalTimeBusy += (this.IsCompletelyBusy ? deltaTime : 0.0f);

this.totalQueueLengthSum += deltaTime \* this.Queue.Count;

this.timeCurrent = value;

foreach (ProcessorModule processor in this.subProcessors)

processor.TimeCurrent = this.timeCurrent;

}

}

internal override sealed void AcceptTask(Task task, IMockWorker? mockWorker)

{

if (this.IsCompletelyBusy)

{

if (!this.Queue.IsFull)

this.Queue.AddLast(task);

else

++this.FailuresCount;

}

else

{

Module? nextModule = this.distribution.GetNextModule(task);

nextModule?.AcceptTask(task, this.mockWorkers[task.CurrentType]);

Console.WriteLine($"|LOG| (TRACE) [{base.Identifier}] sends task to the [{nextModule?.Identifier}]");

this.MoveTimeline(0.0f);

}

}

internal override sealed void CompleteTask()

{

++this.SuccessesCount;

List<ProcessorModule> BusySubProcessors = this.BusySubProcessors;

foreach (ProcessorModule processor in BusySubProcessors)

{

processor.CompleteTask();

if (!this.Queue.IsEmpty)

{

Task newTask = this.Queue.RemoveFirst();

processor.AcceptTask(newTask, this.mockWorkers[newTask.CurrentType]);

Console.WriteLine($"|LOG| (TRACE) [{base.Identifier}] sends task to the [{processor.Identifier}]");

}

}

this.MoveTimeline(0.0f);

}

private protected override sealed void MoveTimeline(float deltaTime)

{

this.TimeNext = this.subProcessors.Min(processor => processor.TimeNext);

}

public override sealed void PrintIntermediateStatistics()

{

Console.Write($"|LOG| (STATS) [{base.Identifier}] ");

Console.WriteLine($"Busy sub-processors: {this.BusySubProcessors.Count}; Queue: {this.Queue.Count}; Successes: {this.SuccessesCount}; Failures: {this.FailuresCount}");

}

public override sealed void PrintFinalStatistics()

{

float busyness = this.totalTimeBusy / this.TimeCurrent;

float queueLengthMean = this.totalQueueLengthSum / this.TimeCurrent;

float failureProbability = (this.SuccessesCount == 0 ? 0 : (float)this.FailuresCount / (this.FailuresCount + this.SuccessesCount));

Console.Write($"|REPORT| [{base.Identifier}] ");

Console.Write($"Busyness: {busyness}; ");

Console.Write($"Queue mean: {queueLengthMean}; ");

Console.Write($"Failures: {this.FailuresCount}; ");

Console.Write($"Successes: {this.SuccessesCount}; ");

Console.Write($"Failure probability: {failureProbability}; ");

Console.WriteLine($"Average busy processors: {this.totalSubProcessorsTimeBusy / this.TimeCurrent}");

foreach (ProcessorModule processorModule in this.subProcessors)

processorModule.PrintFinalStatistics();

}

}