Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Московский государственный технический университет
имени Н.Э. Баумана»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»
КАФЕДРА «Теоретическая информатика и компьютерные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА К КУРСОВОМУ ПРОЕКТУ НА ТЕМУ:

Реализация кооперативной многозадачности и планирование задач в пользовательской среде

Студент ИУ9-52 Зворыгин А.В. _____
(ф.и.о.) (подпись, дата)

Руководитель курсового проекта Вишняков И.Э. _____
(ф.и.о.) (подпись, дата)

Москва, 2020г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В данной работе будет рассмотрена кооперативная многозадачность - эффективный способ использования ресурсов процессора для работы программной системы. Данный способ использования ресурсов компьютера сегодня особенно актуален и используется в распределенных и операционных системах.

Целью этой работы является изучение способов реализации единиц кооперативной многозадачности, а также реализация и анализ способов их планирования.

1 Исследование

В данном разделе будут описаны принципы работы кооперативных многозадачных систем и способы их организации. Также будут рассмотрены способы создания отдельных элементов этих систем и существующие реализации.

1.1 Основные термины

Центральный процессор (англ. CPU) - электронный блок либо интегральная схема, исполняющая машинные инструкции (код программ), главная часть аппаратного обеспечения компьютера или программируемого логического контроллера. Иногда называют микропроцессором или просто процессором. У современных процессоров существуют два основных подхода CISC и RISC [1] [2], самые популярные архитектуры ARM [1] и 8086 [2] соответственно. Различий в реализации для этих архитектур нет, так как реализации такой многозадачности не затрагиваются комплексные операции – операции, которые выполняют сложные функции, такие как работа с регистрами управления или взятие блокировок на кэш-линии [1] [2]. В дальнейшем будут учтены возможные ограничения и возможные потери производительности при реализации. В современных процессорах работают 4, 6 или 8 ядер.

Ядро (англ. Core) – часть процессора – с аппаратной точки зрения представлено как отдельный процессор, зачастую кэш L3 в процессоре является общим среди его ядер. Для примера приводится схема современного процессора архитектуры 8086 – Intel Core i7-3960 X [3] (см. рисунок 1).

Кэш микропроце́ссора — кэш (сверхоперативная память), используемый микропроцессором компьютера для уменьшения среднего времени доступа к компьютерной памяти. Является одним из верхних уровней иерархии памяти.

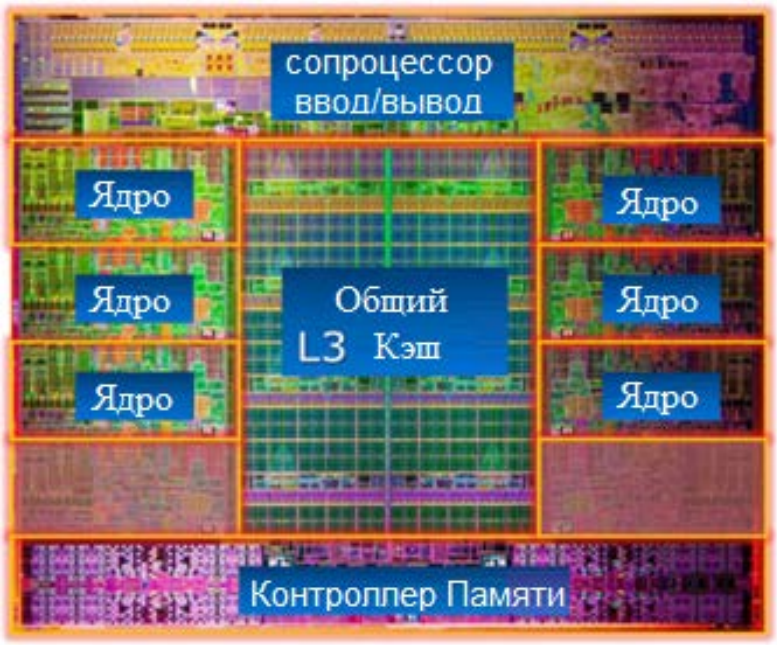

Рисунок 1 - Схема процессора Intel Core i7-3960 X

Каждое ядро использует два уровня кэшей L1 и L2 [4] соответственно. Кэши предназначены для временного хранения памяти и более быстрого доступа к ней [5]. По принципу ближе-меньше-быстрее и дальше-больше-дольше (см. таблица 1) [6].

Таблица 1 – Сравнительная таблица иерархии памяти

	Время доступа	Емкость	Управляется	Находится
Регистры	1 такт	1 Кбайт	Кодом программы	Ядро
L1 Кэш	2-4 такта	32 Кбайт	Аппаратным обеспечением	Чип процессора
L2 Кэш	10 тактов	256 Кбайт		
L3 Кэш	40 тактов	10 Мбайт		
Оперативная память	200 тактов	10 Гбайт	Программным обеспечением	Удаленный чип
Флэш память	10-100 мкс	100 Гбайт		
Жесткий диск	10 мс	1 Тбайт		Механический носитель

Далее будут описаны некоторые компоненты системы Unix, для систем семейства Windows NT механизмы концептуально отличатся не будут. В

операционных системах Unix поддерживающих многозадачность есть механизм многопоточности.

Поток (англ. thread [7] [8] [9]- нить) – логический элемент многозадачной системы, в своем большинстве потоки обладают собственным стеком. Остальные секции (.text, .bss, .data [2] [7]) программы общие. Потоки представляют собой определенную задачу, которую необходимо выполнить и являются логической абстракцией над процессором и его ядрами. Ядро операционной системы предоставляет интерфейс для запуска параллельных потоков на ядрах процессора. Таким образом можно добиться максимальной производительности многоядерного процессора. Стоит отметить, что современные операционные системы выполняют все процессы псевдопараллельно. Это достигается быстрым переключением потоков на ядрах процессора.

В ядрах операционных систем существует очередь задач (англ. run queue) [7] [10] [11] [12], которая упорядочивает потоки, которые необходимо запустить. Какие именно потоки запускать в данный момент и сколько времени на работу давать каждому из них решает планировщик.

Планировщик – модуль ядра системы, который отвечает за эффективное распределение времени между потоками, согласно их приоритетам. Каждый планировщик должен исключать голодание потоков – когда поток получает недостаточно времени для своего исполнения на процессоре.

Переключение потоков дорогостоящая операция. Ее этапы следующие.

1. Вызывается прерывание часов – ядро получает сигнал о произошедшем тике часов.
2. Если количество тиков для определенного потока достаточно – поток прекращает выполнятся на время и готовится уступить место другому потоку.
3. Ядро освобождается от прошлого потока и происходит смена контекста [13] [2] [9] [7]. Все регистры процесса и другая

вспомогательная информация сохраняются, чтобы в последствии вернуть эту информацию обратно.

4. Выставляются регистры процесса нового потока и вспомогательная информация. Новый поток продолжает свою работу.
5. Завершение обработки прерывания.

К тому же во время обработки одного прерывания процессор не может обрабатывать другие прерывания. Также при переключении контекста сбрасываются кэши процессора (в новом контексте новая область памяти), что также влияет на работу в дальнейшем – обращение к памяти будет намного дольше, пока кэши не наполнятся. Вызывание прерывания – самая дорогостоящая операция процессора.

Программа в памяти представляется следующим образом, на рисунке 2 представлены 2 программы – однопоточная – слева и многопоточная – справа.

Рисунок 2 – Память многопоточной и однопоточной программ

Общими для потоков являются текст программы, глобальные переменные, динамическая память, но стек у каждого потока свой.

1.2 Постановка проблемы

Разработчики планировщиков операционных систем ищут способы найти компромисс между интерактивностью и количеством переключений. То есть, идеальный планировщик в Linux (CFR – Complete Fair Scheduler [11]) по своей методологии стремится переключать потоки бесконечно быстро – это идеальная интерактивность. Но в свою очередь бесконечно частая смена контекста заблокирует систему – она будет заниматься только переключением потоков. Поэтому в идеальном планировщике есть минимальные границы времени работы потока на ядре [11].

Планировщик стремится выбрать процесс, который работал меньше всего по времени. Для этого в большинстве моделей планировщиков используется красно-черное дерево [13] [11] для упорядочивания потоков. Планировщик старается взять самую левую вершину дерева – с наименьшим временем. На рисунке 3 представлено красно-черное дерево, сортированное по – виртуальному время работы процесса [11] [7].

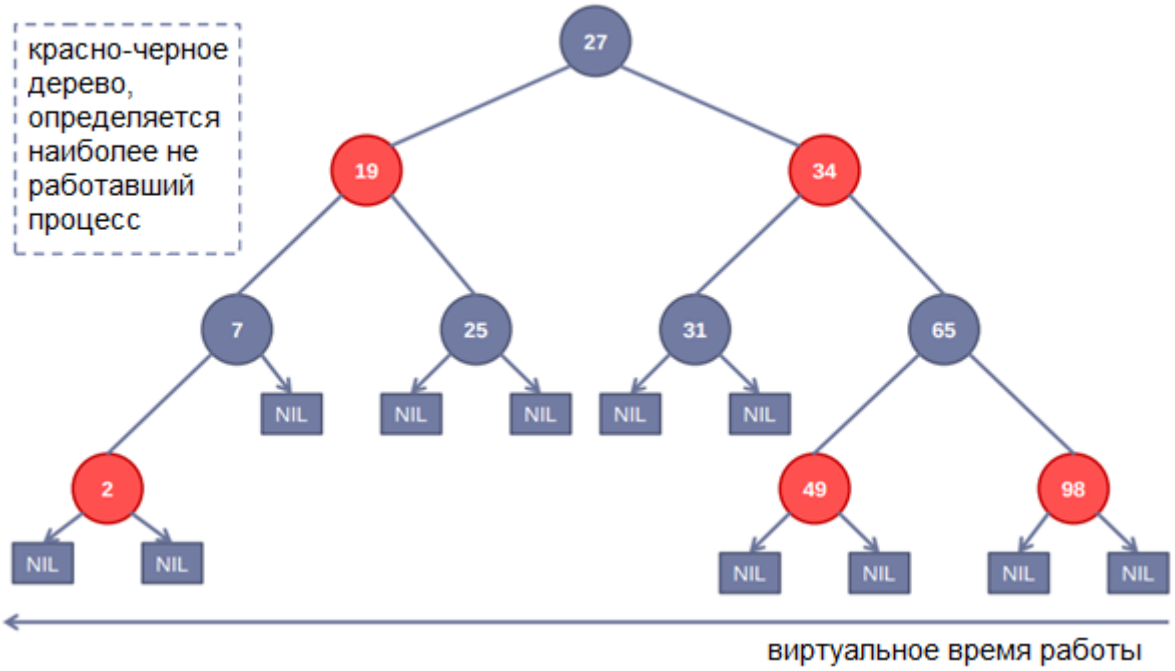

Рисунок 3 – Красно-черное дерево планировщика Linux

Из-за этого существуют большие ограничения на разработчиков ПО. Это основная причина создания кооперативной многозадачности в пользовательской среде.

1.3 Существующие реализации

В данном разделе хотелось бы рассмотреть наиболее удачные реализации кооперативной многозадачности в пользовательской среде. Важно рассмотреть Golang, C++ и Java, как наиболее используемые языки при разработке многозадачных систем. В основном все существующие реализации сводятся к созданию еще одной абстракции – легковесных потоков.

Суть этой абстракции заключается в создании потока, который может быть переключен без использования прерываний. Такого можно достичь если расположить этот логический поток внутри потока операционной системы.

1.3.1 Golang

В качестве примера стоит описать механизм работы горутин. Горутина это логический элемент многозадачной системы. Программа на языке Go в своей

работе стремится выделить столько потоков, сколько у процессора ядер. Так исключается смена контекста между потоками. Горутины выстраиваются в очередь исполнения – аналог очереди задач внутри ядра. Далее планировщик горутин расставляет и снимает горутины на потоках, которые ему выделены(см. рисунок 4) [12] [14].

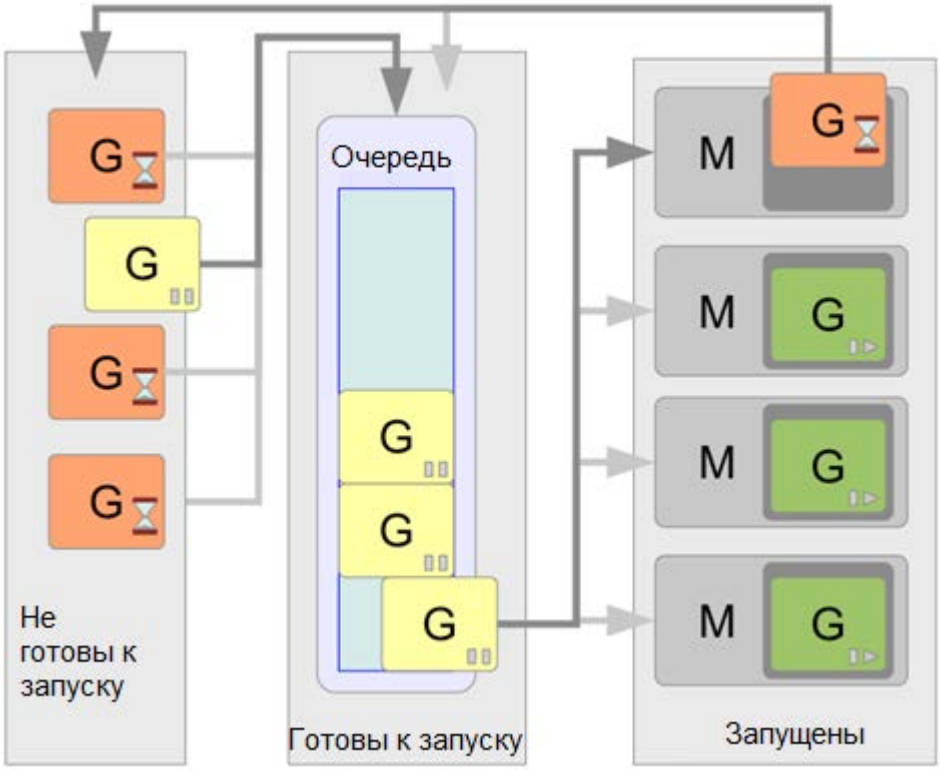

Рисунок 4 – Схема организации многозадачности в Golang

На рисунке 5 представлены схемы двух процессов, первый – работает, используя механизм кооперативной многозадачности, второй используя потоки, которые взаимодействуют с ядром напрямую.

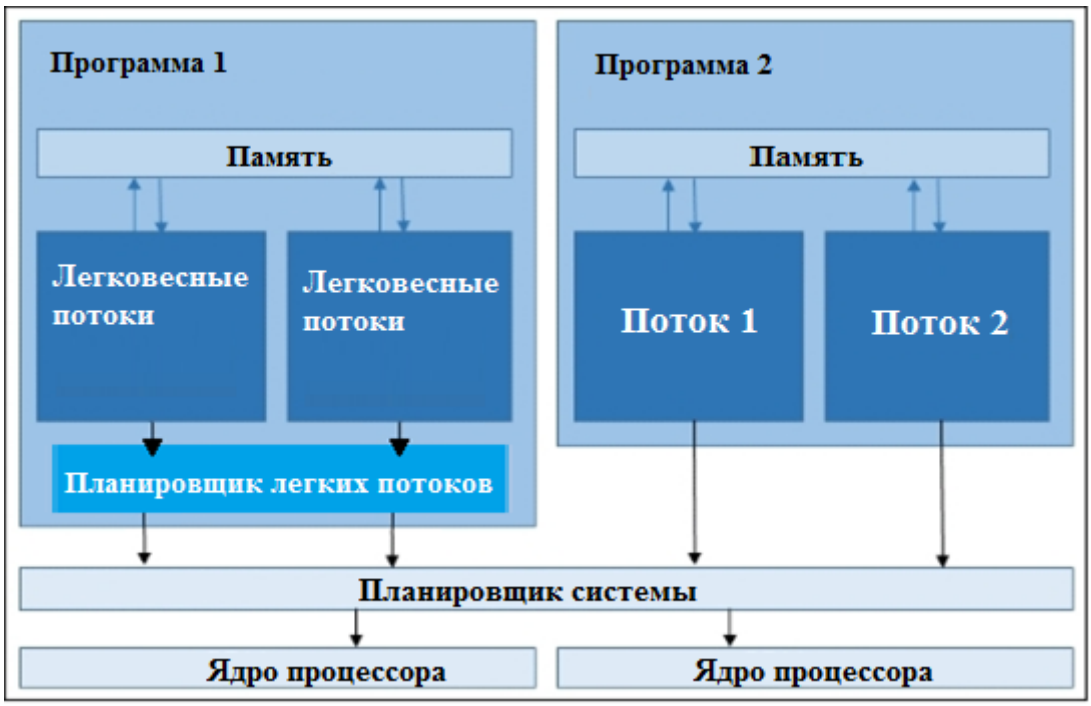

Рисунок 5 – Процесс 1 использующий кооперативную многозадачность и процесс 2 использующий вытесняющую многозадачность

Таким образом организуется планирование многозадачности внутри программы без использования тяжеловесных смен контекста внутри ядра. Поэтому Golang популярен среди разработчиков распределенных, высоконагруженных систем.

1.3.2 Java

Реализация аналога горутин в Java интересна тем, что разработчики языка ввели это на уровне библиотек и улучшения виртуальной машины Java (англ. Java Virtual Machine) [15]. С выходом Java 8 в языке появились элементы кооперативной многозадачности.

1. Легковесный поток –волокно (англ. Fiber) – аналог горутин
2. Очередь легковесных потоков (англ. Fork Join Pull) – аналог планировщика горутин [12]

Разработчики среды разработки Java (англ. Java Runtime Environment) развивают этот механизм, так как он используется в большинстве многозадачных программ, написанных на Java. Переключение контекста

легковесных потоков происходит внутри виртуальной машины Java – что облегчает пользователю понимание работы.

1.3.3 C++

В C++ в стандартных библиотеках такого механизма нет, но в широко известной библиотеке Boost [16] есть реализация сопрограмм (англ. coroutine). Сами по себе они являются аналогом тех же горутин, но в C++ можно выбирать, является ли сопрограмма прерываемой из вне, также механизм предоставляет возможность полного контроля, исследования и расширения механизма, поскольку исходный код находится в открытом доступе.

2 Проектирование

Основной задачей проектирования в данном случае будет определение ключевых интерфейсов элементов кооперативной многозадачности и анализ алгоритмов планирования задач.

2.1 Проектирование единицы многозадачности

Прежде всего необходимо приступить к разработке модели сопрограммы, а также определить необходимы интерфейсы для последующего её использования. Так как эта единица является элементом кооперативной многозадачности, он должна обладать минимум двумя функциями управления:

1. Получить управление (зайти в сопрограммы).
2. Отдать управление (временно выйти из сопрограммы).

Также необходимо определить функции создания и удаления сопрограммы. Поэтому на данный момент можно определить следующий интерфейс.

1. Функция создания сопрограммы принимает предназначенную сопрограмме функцию, которую необходимо исполнить и входные аргументы.
2. Функция освобождения ресурсов сопрограммы работает – освобождает память, которая используется для ее стека.
3. Функция передачи управления передает управление сопрограмме.
4. Функция передачи управления планировщику - позволяет сопрограмме отдать управление.

Также для удобства использования сопрограмм применять локальное хранилище потоков (англ. Thread-Local-Storage) [17] (в качестве глобальных переменных). Таким образом пользователю не нужно явно указывать исполняемую сопрограмму. Это можно сделать поскольку в один момент в одном потоке может исполнятся только одна сопрограмма, поэтому при входе в

сопрограмму в локальное хранилище потока записывается это сопрограмма, а при выходе управление передается обратно - если сопрограмма была вызвана внутри другой сопрограммы, то в локальное хранилище потока будет записана внешняя сопрограмма, иначе будет значение текущей сопрограммы будет пустым. Разумеется, при вызове вне тела сопрограммы произойдет ошибка, так как внутри локального хранилища потока не будет сопрограммы.

Этот элемент многозадачности является простым и его довольно тяжело использовать, потому как необходимо вручную переключатся между сопрограммами и следить за циклическими зависимостями. К тому же использование такого элемента может быть ошибочно, потому что пользователь может использовать блокирующие операции. К примеру, вызовы засыпания потока или чтение буфера, получение ответа с сервера или с клиента. То есть пользователь может вызвать функцию передачи управления до блокирующей операции. Но тогда после возвращения сопрограмма всё равно будет заблокирована. А вызвать функцию передачи управления после входа, например, внутри функции засыпания невозможно. Таким образом, рассмотрев минусы сопрограмм, можно утверждать, что этот элемент является базой для более сложных элементов и не может считаться достаточно эффективным, в первую очередь из-за ручного управления со стороны пользователя.

2.2 Проектирование легковесных потоков

Чтобы решить проблемы, описанные выше, был предложен механизм легковесных потоков, который сочетает в себе и признаки сопрограмм (пользовательская среда и легкая смена контекста) и признаки потоков (возможность прерывания и планирование планировщиком задач). Кооперативная многозадачность, что представлена в Golang, Java и C++, на самом является гибридной и работает поверх потоков, которые предоставляет операционная система. Легковесные потоки обладают следующими свойствами:

1. Работают над потоками операционной системы – легкая смена контекста.

2. Планируются отдельным модулем внутри пользовательской среды.
3. Могут отдавать управление другим легковесным потокам напрямую.

Данные свойства дают существенные преимущества при использовании долгих операций. Как правило — это обращение к базам данных и серверам. Также легковесные потоки позволяют максимально эффективно задействовать ресурсы компьютера при выполнении распределенных задач.

Далее необходимо определить интерфейсы легковесных потоков и их планировщика.

1. Функция создания легковесного потока принимает предназначенную потоку функцию, которую необходимо исполнить и входные аргументы.
2. Функция освобождения ресурсов потока освобождает память, которая используется для его стека.

Стоит заметить, что функции управления легковесными потоками относятся к функциям планировщика.

2.3 Проектирование планировщика

В данном разделе будет описан интерфейс взаимодействия пользователя с планировщиком.

1. Функция создания планировщика - создает новый планировщик с количеством системных потоков, которые он будет использовать.
2. Функция запуска – запускает системные потоки, которые будут обрабатывать легковесные потоки.
3. Функция создания нового потока – добавляет конкретному планировщику новый легковесный поток, принимает функцию и входные аргументы.
4. Функция разделения – создает новый дочерний легковесный поток и добавляет его к планировщику, внутри которого работал родительский легковесный поток.

5. Функция передачи управления – передает управление другому потоку, какому именно выбирает планировщик.
6. Функция засыпания – усыпляет поток на определенное время (в микросекундах), предполагается, что системные функции засыпания также будут поддерживаться, но их использование может проводить к небольшим потерям производительности.
7. Функция синхронизации – блокирует системный поток, пока все легковесные потоки в очереди планировщика не завершатся.
8. Функция слияния потоков – блокирует поток, пока определенный легковесный поток не завершится.
9. Функция завершения работы – удаляет планировщик, предварительно завершив в нем все потоки.

2.4 Анализ алгоритмов планирования задач.

После реализации механизмов кооперативной многозадачности необходимо рассмотреть функции для взаимодействия с очередью потоков, они определяются в конкретной реализации.

1. Функция получения потока – достает поток для исполнения
2. Функция возврата потока – кладет поток обратно в очередь.

Далее будут рассмотрены основные способы определения следующего потока. Разумеется, чтобы добиться максимальной производительности программы, которая использует кооперативную многозадачность необходимо уменьшить количество смен контекста и исключить задержки на вызове синхронизации потоков друг с другом. Рассмотрим пример – сортировку слиянием.

1. Рекурсивно каждому потоку выделяется подмассив, который необходимо отсортировать.
2. Таким образом, получается строго сбалансированное бинарное дерево, листьями такого дерева являются потоки, выполняющие сравнение 2 элементов.

3. Все узлы собирают результаты дочерних потоков и передают их родительским потокам – в каждом узле будет синхронизация дочерних потоков – пока они не выполнятся родительский поток не может работать.

Таким образом получим дерево вызовов на рисунке 6.

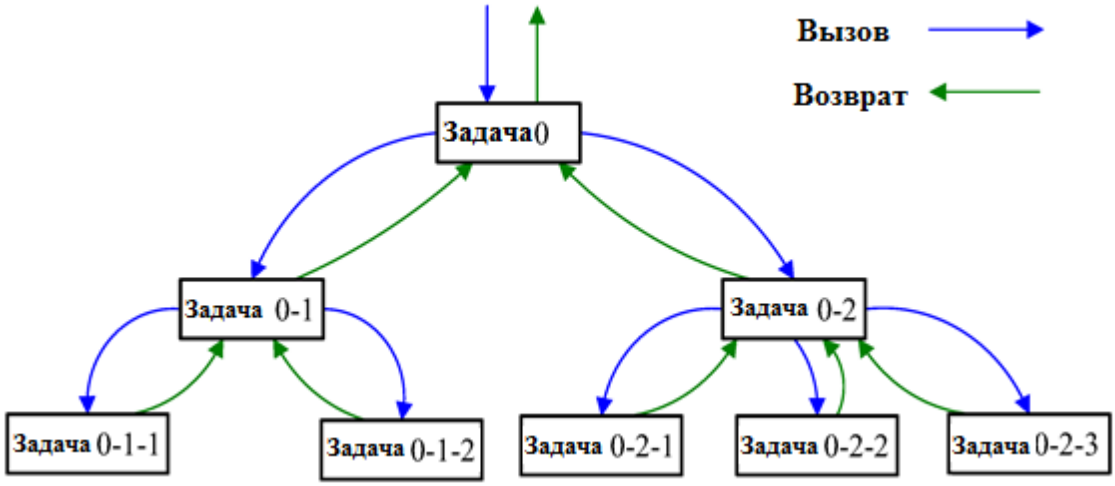

Рисунок 6 – Дерево вызовов подзадач

Тогда худший случай — это путь до листа, в котором каждый узел обладает лишь одним не выполненным элементом. Например, крайний левый путь выполнен не был, а все остальные вершины были выполнены (см. рисунок 7) (красным помечены задачи, которые уже были выполнены). Вызов Задача 0-1 не сможет завершится, пока не завершатся вызовы Задача 0-1-1 и Задача 0-1-2. Таким образом при любом количестве системных потоков, которые можно задействовать, будет работать только 1, который и выполнит этот путь рекурсивно (Задача 0-1-1 -> Задача 0-1 -> Задача 0).

У дочерних вызовов должен быть больший приоритет, нежели у родительских вызовов. Таким образом исключается ожидание на синхронизации родительских потоков с дочерними.

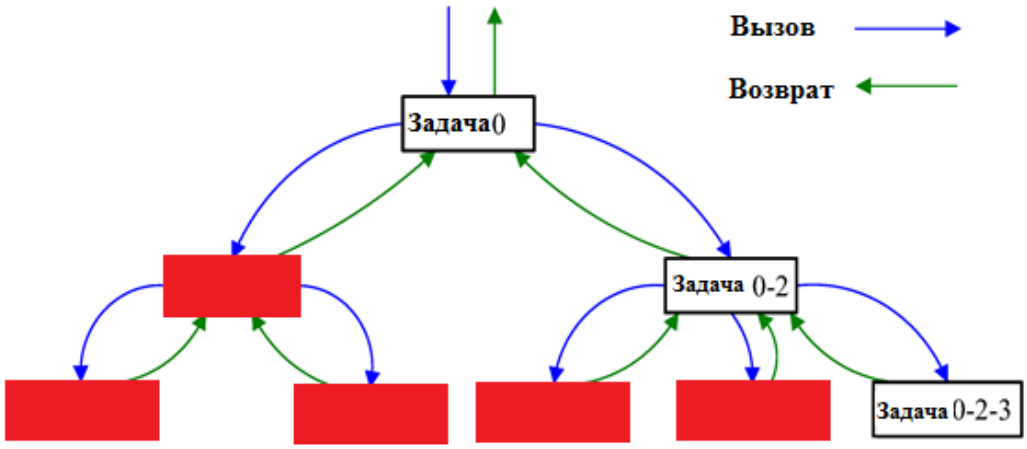

Рисунок 7 – Худший случай планирования подзадач

Для определения приоритетов в очереди легковесных потоков необходимо использовать соответствующие структуры данных. Необходимо поддерживать приоритет потоков по времени работы в потоке, если у них совпадает уровень в дереве вызовов. Таким образом будет исключатся их голодание. Далее приводится список структур данных, которые помогут добиться выполнения необходимых требований.

1. Красно-черное дерево (англ. Red-Black Tree) [13]
2. Фибонначиева куча (англ. Fibonacci Heap) [18] [19]
3. Тонкая куча (англ. Thin Heap) [19]
4. Усеченное-дерево (англ. Splay Tree) [10]

Данные структуры обладают возможностью извлечения минимального элемента и известны как наиболее эффективные в своем роде. Далее будут коротко описаны причины выбора этих алгоритмов и приведено их краткое описание. Разумеется, данные структуры данных должны быть потоко-безопасными, либо вызовы к ним должны быть защищены блокировкой. Синхронизация на структуре данных может тратить процессорное время (ожидание на захват блокировки), поэтому в лучшем случае данные структуры должны быть потоко-безопасными и работать корректно без дополнительных синхронизаций, иначе говоря, принадлежать к классу неблокирующих алгоритмов. Чтобы доказать верность предложенной теории, также необходимо реализовать классическую очередь, которая не поддерживает приоритетность

элементов. Результаты, полученные для предложенных структур данных и с использованием приоритетности задач, будут сравниваться её результатами. Таким образом получится на практике доказать эффективность такого метода организации планирования.

Таблица 2 – Сравнение асимптотических сложностей алгоритмов

	Вставка	Удаление минимального элемента
Фибонначиева куча	Амортизированное O(1)	O(log n)
Тонкая куча	Амортизированное O(1)	O(log n)
Красно-черное дерево	O(log n)	O(log n)
Косое дерево	O(log n)	O(log n)

Наиболее эффективными структурами предположительно будут тонкая и фибонначиева кучи.

2.4.1 Красно-черное дерево

Красно-черное дерево – схематично представлено на рисунке 8, это наиболее популярное сбалансированное двоичное дерево поиска. Оно используется в планировщике ядер систем Linux и BSD. Обладает хорошей асимптотической сложностью и довольно низкой константой. Данная структура данных несколько устарела и на смену ей приходят более эффективные алгоритмы. АВЛ дерево не рассматривается, поскольку в нем происходят повороты на каждом изменении дерева и несмотря на асимптотическое превосходство на практике обладает худшими результатами.

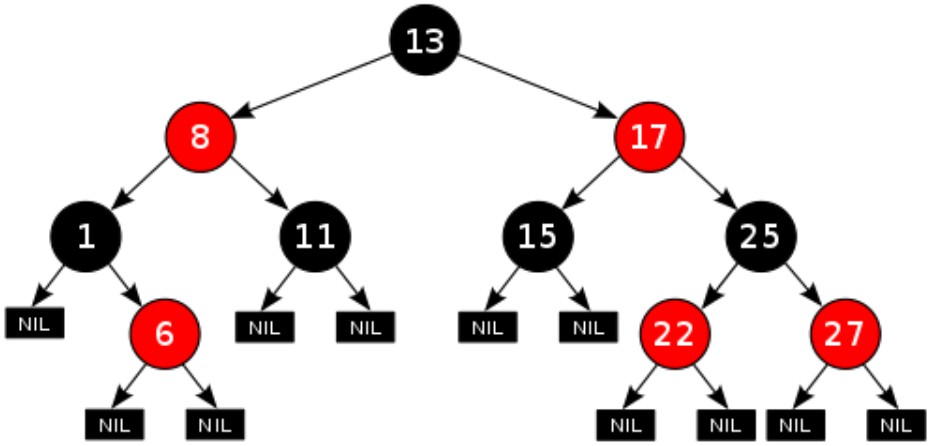

Рисунок 8 - Красно-черное дерево

2.4.2 Фибонначиева и тонкая куча

Тонка куча наиболее эффективная, как по памяти, так и по скорости среди других структур данного класса, также часто используется в практических задачах.

Эта структура, наиболее подходящая для планировщика, если нет необходимости итерироваться по элементам внутри структуры. Фибонначиева куча отличается принципиальным подходом и большими константами.

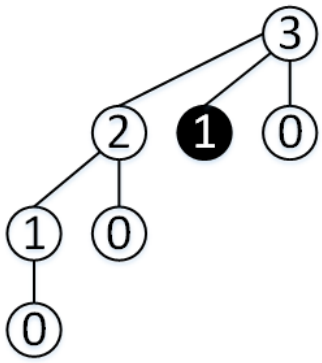

Рисунок 9 - Тонкая куча

2.4.3 Косое дерево

Данное дерево примечательно тем, что при практическом показывает результаты, превосходящие все другие деревья поиска. Это достигается за счет неполной сбалансированности дерева. Это является серьезным минусом для real-

time систем, в частности именно из-за меняющейся на ходу асимптотики от этой структуры отказались разработчики ядра Linux.

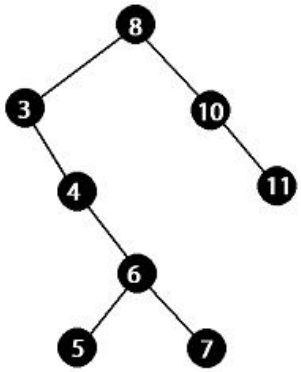

Рисунок 10 – Косое-дерево

3 Реализация

В данном разделе будет описана реализация смены контекста и некоторых важных функций.

3.1 Выбор используемых технологий

В качестве языка программирования для реализации этого механизма был выбран язык C (GNU 99 [20]) и диалект ассемблера NASM [21], как наиболее эффективные для системного программирования. Также причиной выбора являлось то, что механизмы, которые можно реализовать на языке C можно в последствии без труда перенести на другие языки, так как этот язык содержит минимум абстракций на уровне языка. Также язык C позволяет организовать удобное взаимодействие с кодом на ассемблере. В качестве утилит для сборки проекта был выбран CMake [22] [23], как наиболее удобная утилита для создания Makefile'ов. Репозиторий с исходным кодом хранится на аккаунте сервиса GitHub [24]. Для создания графиков будет использоваться Python 3 и библиотеки для построения – Numpy [25], Matplotlib [26].

3.2. Создание рабочей директории

Сначала необходимо определится со структурой проекта. Будет использоваться стандартная схема для проектов, написанных на C\C++.

1. images – построенные графики, показывающие результаты тестирования
2. include – папка для заголовочных файлов
3. src - папка для файлов исходного кода
4. tests - папка содержащая тесты и бенчмарки проекта
5. build.sh содержит скрипт для сборки проекта.
6. run.sh содержит скрипт для запуска тестов проекта
7. script.py содержит код для создания графиков проекта
8. CMakeLists.txt корневой файл для CMake

Далее в каждой подпапке папки src будет содержатся CMakeLists.txt, который будет управлять сборкой отдельных компонент.

3.3 Реализация смены контекста

Чтобы приступить к реализации смены контекста легковесных потоков, необходимо детально рассмотреть механизм работы смены контекста в потоках, которые предоставляет операционная система. При переключении контекста происходит сохранение и восстановление следующей информации [10] [2] [5] [27]:

1. Контекст регистров общего назначения и флаговых регистров
2. Контекст сопроцессора для операций с плавающей точкой
3. Состояние некоторых управляющих регистров
4. Регистры состояния специфичные для разных архитектур

Далее необходимо рассмотреть способ смены контекста внутри одного потока. Для начала рассмотрим смену стековых кадров при вызове функции. Рассмотрим стек вызовов определенного потока. Для вызова функции необходимо совершить вызов call и передать в качестве переменной адрес функции. Но для того, чтобы вернуться обратно вызову ret необходимо иметь адрес возврата. Функция call (см. листинг 1) кладет на стек адрес возврата – текущий указатель на стек (англ. stack pointer – rsp) [4] [2] [11]. Далее на стек помещается указатель на начало стека (англ. base pointer – rbp) [4] [2] [11]. Указатель на стек и указатель на текущую инструкцию (англ. instruction pointer rip) [4] [2] [11] инкрементируются автоматически (см. рисунок 11).

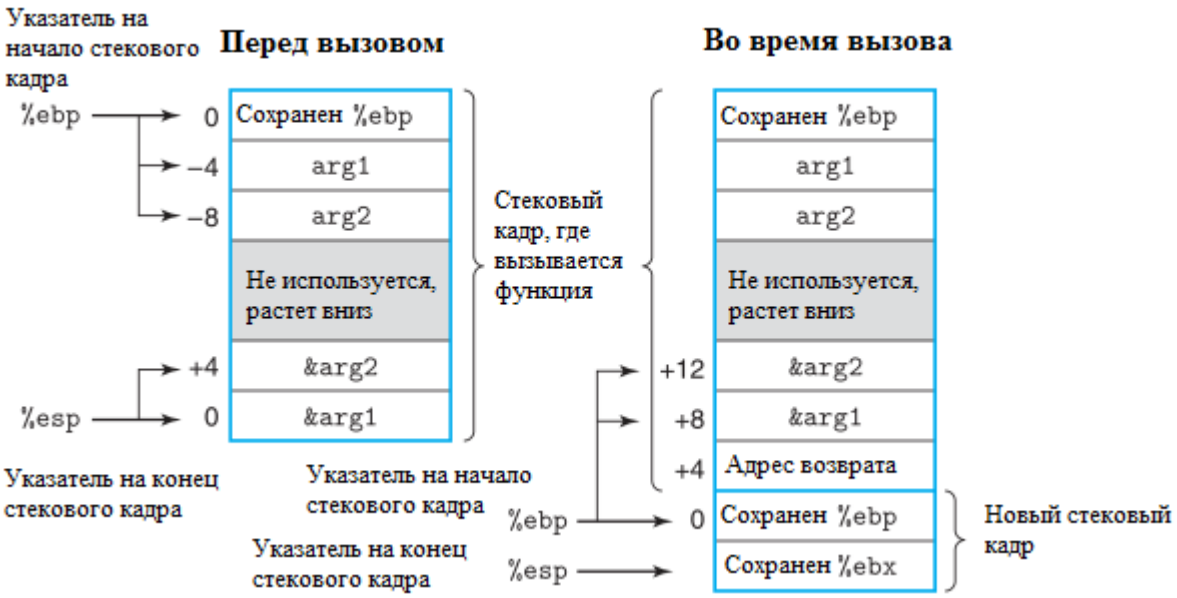

Рисунок 11 – Схема стека при вызове функции

Для того чтобы восстановить указатель на начало стека [11] [2] [4] для предыдущего стекового кадра при возврате из функции, в начале каждого кадра лежит указатель на предыдущий. Таким образом реализуется работа стековых кадров и этот принцип можно использовать для реализации смены контекста в одном потоке.

Листинг 1 – Операция call схематично

```
1.  pushq %rbp
2.  * Сохранение текущего указателя на начало кадра,
3.  * при выходе из функции он будет восстановлен
4.
5.  movq %rsp, %rbp
6.  * Перемещение указателя на начало кадра
7.  * на место текущего указателя на вершину кадра
```

Тогда вызов ret делает всё зеркально (см. листинг 2).

Листинг 2 – Операция ret схематично

```
1.  movq %rbp, %rsp
2.  * Откат указателя на вершину кадра
3.
4.  pop %rbp
5.  * Восстанавливается указатель на начало кадра
```

Можно считать, смена контекста — это некий вызов функции, при котором происходит сохранение состояния и прыжок в другой контекст. Рассмотрим

функцию смены контекста (см. листинг 3). Она будет принимать два аргумента. Каждый из них это указатель на начало стекового кадра. Тогда входные параметры находятся в регистрах rdi и rsi [4] (первый и второй соответственно).

Листинг 3 – Операция смены стека

```
1.  movq %rsp, %(rdi)
2.  * Сохранение текущего stack pointer в переменную,
3.  * из которой потом можно восстановить обратно
4.
5.  movq (%rsi), %rsp
6.  * Изменение stack pointer,
7.  * после этого момента стек изменен
```

В процессорах 8086 каждая функция обязуется не менять некоторые регистры родительской функции. А именно rbp, rbx, r12, r13, r14, r15 [4]. Поэтому необходимо их сохранить. А сохранить их можно положив на стек перед сменой контекста и восстановить после. Тогда конечный код смены контекста будет выглядеть следующим образом (см. листинг 4):

Листинг 4 – Операция смены контекста

```
 1.   pushq %r15
 2.   pushq %r14
 3.   pushq %r13
 4.   pushq %r12
 5.
 6.   pushq %rbx
 7.   pushq %rbp
 8.
 9.   * Сохранение на стек текущих регистров
10.
11.   movq %rsp, (%rdi)
12.   movq (%rsi), %rsp
13.
14.   * Снятие со стека регистров нового потока
15.
16.   popq %rbp
17.   popq %rbx
18.
19.   popq %r12
20.   popq %r13
21.   popq %r14
22.   popq %r15
23.
24.   retq
```

Отметим, что сохранять указатель на инструкцию не требуется, потому что он будет сохранен автоматически – аналогично работе функций. Далее необходимо определить базу индукции. Необходимо загрузить необходимые регистры заранее – при создании легковесного потока.

3.4 Реализация сопрограммы

Для реализации сопрограммы ей в первую очередь необходим собственный стек. Рассмотрим функцию создания сопрограммы (см. приложение А Функция создания и разметки стека легковесного потока). При вызове функции в нее передается указатель на сопрограмму, вызываемая функция и аргументы, которые ей передаются. Для стека сопрограммы выделяется память с помощью функции mmap [28]. Затем выделяются определенные права для данной памяти функцией protect. Затем находится вершина стека – это самый большой адрес в выделенной памяти. Далее стек

смещается так, чтобы начало стека было кратно 16. Затем происходит выделение памяти под сохраненный контекст – stack saved context. После этого заранее сохраняется функция – трамплин (см. листинг 5) в указатель на инструкцию, эта функция которая будет вызвана при первом прыжке в контекст сопрограммы и из нее можно вернуться обратно во внешний контекст. После того как контекст был создан, сохраняется указатель на него, который будет использоваться при первом прыжке в контекст.

Листинг 5 – Функция обратного вызова – трамплин

```
1.  static void trampoline ()
2.  {
3.      current_coroutine->routine (
4.                      current_coroutine->args);
5.      suspend ();
6.  }
```

В current_coroutine хранится текущая исполняющаяся горутина для данного потока – то есть в локальном хранилище потоков. А функция передачи управления (см. листинг 6) передает управление внешнему контексту исполнения.

Листинг 6 – Функция передачи управления внешнему контексту

```
1.  void suspend ()
2.  {
3.      coroutine *next_coroutine = current_coroutine;
4.      current_coroutine =
5.                      next_coroutine->external_routine;
6.      switch_context(&next_coroutine->routine_context,
7.                      &next_coroutine->caller_context);
8.  }
```

Тогда необходимо описать возвращение в сопрограмму после выхода из нее – это функция продолжения работы (см. листинг 7).

Листинг 7 – Функция передачи управления сопрограмме

```
1.  void resume (coroutine *coroutine_by_resume)
2.  {
3.      current_coroutine = coroutine_by_resume;
4.
5.      switch_context (
6.            &coroutine_by_resume->caller_context,
7.            &coroutine_by_resume->routine_context);
8.  }
```

На основе разработанной сопрограммы далее необходимо разработать легковесные потоки и планировщик для управления ими.

3.5 Разработка планировщика и потоков

Самой главной функцией планировщика будет функция бесконечного цикла – цикла планирования потоков (см. в приложение А Функция планирования задач). Её суть заключается в планировании потоков если они находятся в очереди, иначе поток, обрабатывающий задачи, засыпает, чтобы не тратить процессорное время. Функция scheduler_pause блокирует планирование пока планировщик не начал свою работу или остановлен. Если планировщик будет удален, поток завершает свою работу. Функция get_from_pool неопределенна в главном заголовочном файле, а определяется отдельно для каждой реализации. Прежде чем перейти к другим функциям стоит определить возможные состояния потоков (см. листинг 8).

Листинг 8 – Перечисление состояний потоков

```
1.  enum fiber_state
2.  {
3.      STARTING,  // Состояние до запуска потока
4.      RUNNABLE,  // Состояние незаконченного потока
5.      RUNNING,   // Состояние во время работы
6.      SLEEPING,  // Состояние, когда поток спит
7.      TERMINATED // Состояние после завершения потока
8.  };
```

Далее стоит рассмотреть функцию run_task (см. в приложение А Функция запуска задач). В этой функции происходит возврат в контекст потока (строка 8). После возвращения из контекста потока происходит проверка, был ли поток

завершен. Если поток завершился, то освобождается память, выделенную ему. Иначе поток возвращается в очередь. Функция, возвращающая поток в очередь, определяется в конкретной реализации. Определим функцию трамплин для потока (см. листинг 9) – она отличается от трамплина сопрограмм.

Листинг 9 – Функция трамплина для потока

```
 1.  static void fiber_trampoline ()
 2.  {
 3.      fiber *temp = current_fiber;
 4.
 5.      temp->state = RUNNING;
 6.      temp->routine(temp->args);
 7.      temp = current_fiber;
 8.
 9.      temp->state = TERMINATED;
10.
11.      switch_context(&temp->context,
12.                     &temp->external_context);
13.  }
```

Рассмотрим функции (см. листинги 10 и 11) смены контекста потока на контекст планировщика.

Листинг 10 – Функция передачи управления планировщику

```
1.  void yield()
2.  {
3.       fiber *temp = current_fiber;
4.       temp->state = RUNNABLE;
5.       switch_context(&temp->context,
6.                      &temp->external_context);
7.  }
```

Листинг 11 – Функция засыпания потока на несколько микросекунд

```
1.  void sleep_for (unsigned long duration)
2.  {
3.      fiber *temp = current_fiber;
4.
5.      temp->wakeup = clock ();
6.      temp->wakeup +=
7.           clock_to_microseconds((long)duration);
8.
9.      temp->state = SLEEPING;
10.     switch_context(&temp->context,
11.                    &temp->external_context);
12. }
```

4 Тестирование

Для проведения комплексного тестирования прежде всего необходимо реализовать юнит-тестирование, стресс-тестирование для проверки работоспособности разработанных модулей. Также необходимо разработать бенчмарки для сбора метрик для реализованных алгоритмов планирования. По полученным результатам.

4.1 Модульное тестирование

Модульное тестирование необходимо для проверки корректной работоспособности сопрограмм, потоков и планировщика для одного потока. При таком тестировании исключаются возможные многопоточные проблемы и проверяются базовые элементы.

4.2 Стресс-тестирование

Стресс-тестирование необходимо для отслеживания ошибок работы в многопоточной среде. Для комплексного стресс-тестирования были написаны необходимые стресс-тесты, для каждого варианта алгоритма планирования.

4.3 Нагрузочное тестирование

Были реализованы различные сценарии использования планировщика. Были рассмотрены худшие случаи планирования задач, такие как частое принудительное переключение контекста – использование методов передачи управления и засыпания и использование рекурсивной синхронизации.

4.4 Анализ результатов

Для проведения анализа результатов был реализован модуль на языке Python для построения графиков, с использованием библиотек Pandas и MatplotLib.

4.4.1 Тестирование рекурсивных синхронизаций

Здесь будет протестировано поведение планировщика при частых рекурсивных синхронизациях. Ожидается, что благодаря ранжированию потоков по их уровням, количество рекурсивных синхронизаций и как следствие синхронизаций между системными потоками уменьшится по сравнению с обычной очередью.

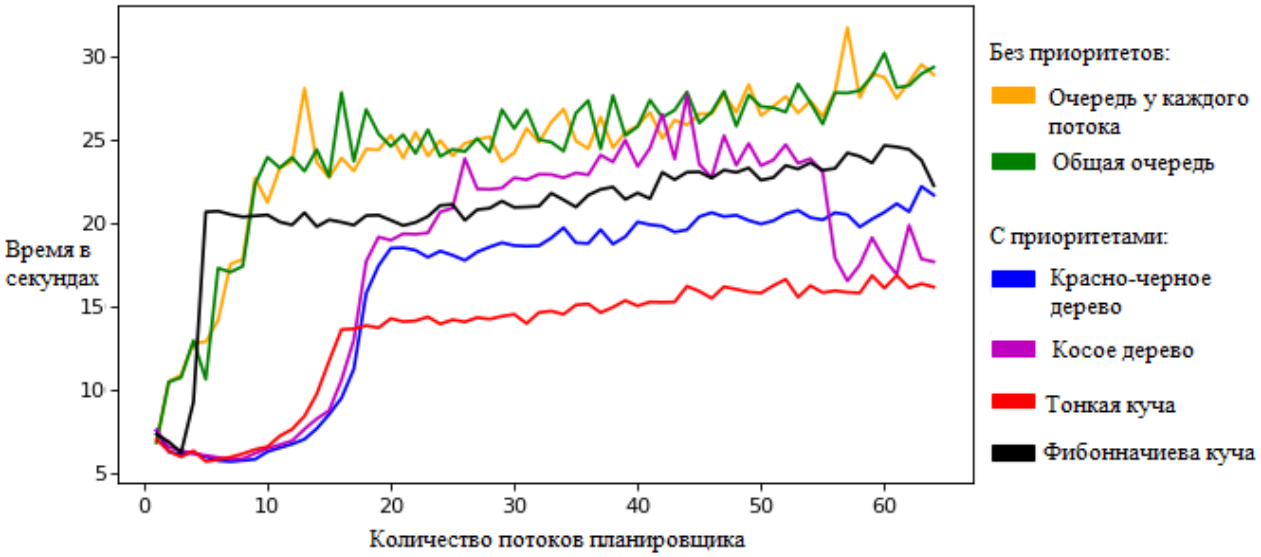

Рисунок 12 - Тестирование с рекусивными синхронизациями

Был получен ожидаемый результат (см. рисунок 12). Алгоритмы с ранжированием потоков отработали за меньшее время. Стоит заметить, что фибонначиева куча несмотря на хорошие асимптотики проигрывает сбалансированным деревьям с более плохими асимптотиками. Лучшим вариантов в данном тесте является тонкая куча. Также заметно, что за счет неполной сбалансированности косого дерева результаты не стабильны, если количество задач меньше по сравнению с количеством потоков.

4.4.2 Тестирование с использованием функций засыпания

В данном тесте проверяется работа планировщика при частых усыплениях потоков. Это значит, в алгоритме, где используется очередь, планировщику будут постоянно встречаться потоки, которые еще спят. Таким образом время работы очереди будет больше.

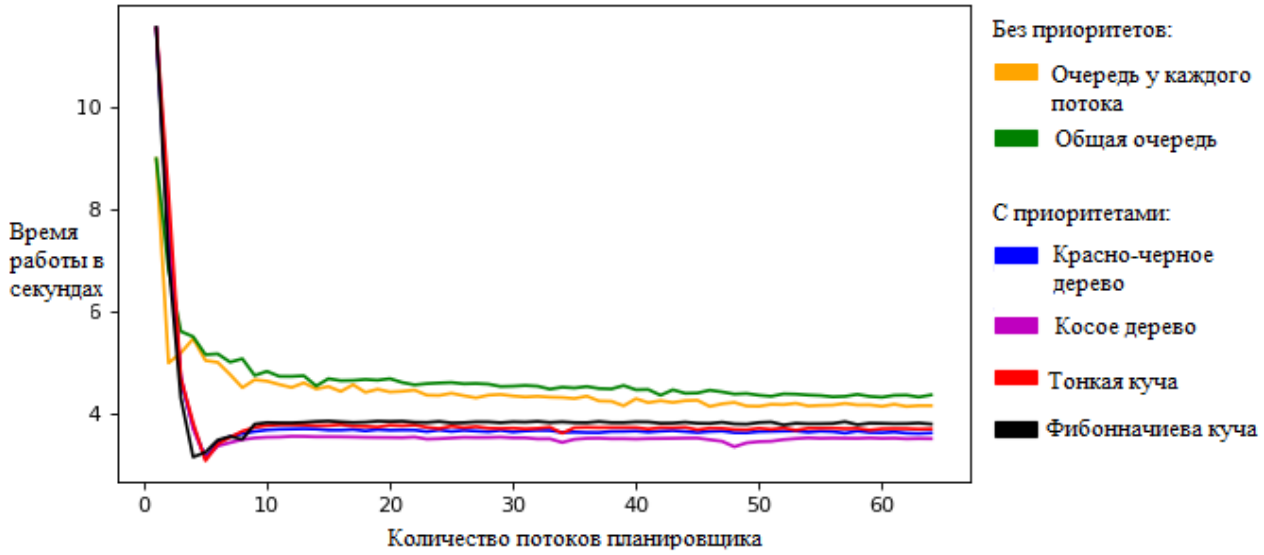

Рисунок 13 - Тестирование вызовом функций засыпания

На рисунке 13 можно видеть, что предположение было верным – время работы очередей заметно больше.

4.5 Вывод

В общем случае распараллеливания задач с помощью потоков, с выполнением тяжеловесных операций, таких как обращение к базе данных, и рекурсивными синхронизациями потоков тонкая куча является наиболее эффективным алгоритмом.

ЗАКЛЮЧЕНИЕ

Были рассмотрены способы реализации кооперативной многозадачности и её элементов. Был определен наиболее эффективный способ планирования задач – использование в качестве очереди потоков тонкой кучи с сортировкой по вложенности и отработанному времени.

Разумеется, в данной работе есть недостатки: не до конца понятно поведение некоторых структур данных в качестве очереди потоков и существуют ли более эффективные способы планирования. Изучение этих вопросов будет продолжено.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 Stokes J. RISC and CISC, Side by Side // Ars Technical. 1999.

2 Corporation I. 4.10.1 Process-Context Identifiers (PCIDs). Vol 3A: System Programming Guide, Part 1. // In: Intel 64 and IA-32 Architectures Software Developer's Manual. 2016.

3 Intels-Core-i7-3960x-processor [Электронный ресурс] // Techreport: [сайт]. URL: https://techreport.com/review/21987/intels-core-i7-3960x-processor (дата обращения: 28.07.2020).

4 X86 64 Register and Instruction Quick Start [Электронный ресурс] // The Seneca Centre for Development of Open Technology: [сайт]. URL: https:// wiki.cdot.senecacollege.ca/wiki/X86_64_Register_and_Instruction_Quick_Start (дата обращения: 07.08.2020).

5 Knuth D.E. The Art of Computer Programming 3rd Edition. Vol 1 Fundamental Algorithms. Redwood-City: Addison Wesley Longman Publishing Co, 1998.

6 Toy E., Wing N. Computer hardware/software architecture. Prentice-Hall: Englewood Cliffs, N.J. : Prentice-Hall, 1986.

7 Butenhof D.R. Programming with POSIX Threads, 1997.

8 Kai L. Interaction Between the User and Kernel Space in Linux, Technische Universität, Berlin, 2017.

9 Marshall K. A New Virtual Memory Implementation for Berkeley UNIX, University of California, Berkeley, 1986.

10 Albers S., Karpinski M. Randomized Splay Trees Theoretical and Experimental Results. 2002.

11 Reyes C.M., González I.L. Optimizing The Linux Scheduler For Performance, 2017.

12 The Go Programming Language Specification [Электронный ресурс] // Golang: [сайт]. URL: https://golang.org/ref/spec#Type_assertions (дата обращения: 03.08.2020).

13 Red–Black Trees // In: Introduction to Algorithms (second ed.) / Ed. by Cormen T., Herlihy T. 2001.

14 Lamport L.M., Lasley D.T. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs // Transactions on Computers. 1979.

15 Lea D. A Java Fork/Join Framework, State University of New York at Oswego,.

16 Documentation Boost [Электронный ресурс] // Boost: [сайт]. URL: https://www.boost.org/doc/ (дата обращения: 05.08.2020).

17 Thread Local Storage [Электронный ресурс] // Oracle Docs: [сайт]. URL: https://docs.oracle.com/cd/E19683-01/817-3677/chapter8-1/index.html (дата обращения: 05.08.2020).

18 Fibonacci Heaps // In: Algorithms and Data Structures: The Basic Toolbox / Ed. by Mehlhorn E., Kurt C., Sanders F. Springer, 2008.

19 Thin Heaps // In: Thick Heaps / Ed. by Haim Y., Tarjan E., Kaplan S. 2006.

20 Language Standards Supported by GCC [Электронный ресурс] // GCC, the GNU Compiler Collection: [сайт]. URL: https://gcc.gnu.org/onlinedocs/gcc-3.4.2/gcc/Standards.html (дата обращения: 05.08.2020).

21 Nasm Into [Электронный ресурс] // NASM: [сайт]. URL: https://nasm.us/ (дата обращения: 05.08.2020).

22 CMake Intro [Электронный ресурс] // CMake: [сайт]. URL: https://cmake.org/ (дата обращения: 05.08.2020).

23 CMake Tutorial [Электронный ресурс] // CMake.org: [сайт]. URL: https://cmake.org/cmake/help/latest/guide/tutorial/index.html (дата обращения: 06.12.2020).

24 Repository [Электронный ресурс] // GitHub: [сайт]. URL: https://github.com/don-dron/scheduler (дата обращения: 17.12.2020).

25 Numpy Docs [Электронный ресурс] // Numpy: [сайт]. URL: https://numpy.org/ (дата обращения: 05.08.2020).

26 Matplotlib Docs [Электронный ресурс] // Matplotlib: [сайт]. URL: https://matplotlib.org/ (дата обращения: 05.08.2020).

27 Language Design in the Service of Software Engineering [Электронный ресурс] // The Go Programming Language: [сайт]. URL: https://talks.golang.org/2012/splash.article (дата обращения: 03.08.2020).

28 mmap(2) — Linux manual page [Электронный ресурс] // Linux man pages: [сайт]. URL: https://man7.org/linux/man-pages/man2/mmap.2.html (дата обращения: 07.08.2020).

29 Fiber Pool The framework for maximum performance [Электронный ресурс] // TrinkMeta: [сайт]. URL: http://www.thinkmeta.de/en/fiberpool_overview.html (дата обращения: 28.07.2020).

30 Microsoft Docs [Электронный ресурс] // Fibers: [сайт]. URL: https://docs.microsoft.com/ru-ru/windows/win32/procthread/fibers?redirectedfrom=MSDN (дата обращения: 28.07.2020).

31 Future [Электронный ресурс] // Oracle Docs: [сайт]. URL: https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html (дата обращения: 28.07.2020).

32 Linux Kernel [Электронный ресурс] // Scheduler Design CFR: [сайт]. URL: https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html (дата обращения: 28.07.2020).

33 LWN [Электронный ресурс] // CPU caches: [сайт]. URL: https://lwn.net/Articles/252125/ (дата обращения: 28.07.2020).

34 Faster Translations // In: Operating Systems: Three Easy Pieces / Ed. by Remzi H. 2014.

35 RISC and CISC Processors [Электронный ресурс] // Study to night: [сайт]. URL: https://www.studytonight.com/computer-architecture/risc-cisc-processors (дата обращения: 28.07.2020).

36 Keller R.M. Some theoretical aspects of applicative multiprocessing // International Symposium on Mathematical Foundations of Computer Science. Lake City. 2005.

ПРИЛОЖЕНИЯ

ПРИЛОЖЕНИЕ А.

Функция создания и разметки стека легковесного потока.

```
1.  static int setup(coroutine *coroutine_, void
2.                           (*trampoline)(void *))
3.  {
4.          void *start = mmap(/*addr=*/0,
5.                          /*length=*/STACK_SIZE,
6.                          /*prot=*/PROT_READ |
7.                                   PROT_WRITE,
8.                          /*flags=*/MAP_PRIVATE | 0x20,
9.                          /*fd=*/-1, /*offset=*/0);
10.
11.     int ret = mprotect(/*addr=*/(void *)
12.          ((size_t)start + pages_to_bytes(4)),
13.                         /*len=*/
14.                         pages_to_bytes(4),
15.                         /*prot=*/PROT_NONE);
16.
17.     if (ret)
18.     {
19.         munmap(start, STACK_SIZE);
20.         return ret;
21.     }
22.
23.     stack_builder stackBuilder;
24.     stackBuilder.top = (char *)
25.                ((size_t)start + STACK_SIZE - 1);
26.     stackBuilder.word_size = sizeof(void *);
27.
28.     align_next_push(&stackBuilder, 16);
29.     allocate(&stackBuilder,
30.                    sizeof(stack_saved_context));
31.
32.     stack_saved_context *saved_context =
33.         (stack_saved_context *)stackBuilder.top;
34.
35.     saved_context->rip = (void *)trampoline;
36.     coroutine_->routine_context.rsp =
37.                                     saved_context;
38.
39.     coroutine_->stack = start;
40.
41.     return 0;
42. }
43.
44. int create_coroutine(coroutine *new_coroutine,
45.                  void (*routine)(), void *args)
```

```
{
    new_coroutine->routine = routine;
    new_coroutine->complete = 0;
    new_coroutine->args = args;
    int ret = setup(new_coroutine, trampoline);

    if (ret)
    {
        return ret;
    }
    new_coroutine->external_routine =
                                current_coroutine;
    return 0;
}
```

Функция планирования задач.

```
static void schedule()
{
    while (1)
    {
        scheduler_pause();

        if (current_scheduler->terminate)
        {
            return;
    }

    fiber *fib = get_from_pool();

    if (fib)
    {
        run_task(fib);
    }
    else
    {
        usleep(1000);
    }
}
```

Функция запуска задач.

```
static void run_task(fiber *routine)
{
        current_fiber = routine;
        fiber *temp = current_fiber;

        current_fiber->state = RUNNING;

        switch_context(
            &current_fiber->external_context,
        &current_fiber->context);

    if (current_fiber->state != TERMINATED)
```

```
13.       {
14.           return_to_pool(current_scheduler,
15.                          current_fiber);
16.           current_fiber = NULL;
17.       }
18.       else
19.       {
20.           current_fiber = NULL;
21.           inc((unsigned long *)
22.             &current_scheduler->end_count);
23.
24.           free_fiber(temp);
25.       }
26.   }
```