Министерство науки и высшего образования Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

”Национальный исследовательский университет ИТМО“

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

РАЗРАБОТКА И РЕАЛИЗАЦИЯ МЕТОДОВ ЭФФЕКТИВНОГО ВЗАИМОДЕЙСТВИЯ ПРОЦЕССОВ В РАСПРЕДЕЛЕННЫХ СИСТЕМАХ

Автор Губарев Владимир Юрьевич _____

Направление подготовки 09.04.04 Программная инженерия

Квалификация магистр

Руководитель Косяков М.С., к.т.н. _____

Санкт-Петербург, 2020 г.

Обучающийся Губарев Владимир Юрьевич
Группа Р42111 Факультет/институт/кластер ПИиКТ

Направленность (профиль), специализация
Информационно-вычислительные системы, Интеллектуальные системы

ВКР принята «_____» _____ 20___ г.

Оригинальность ВКР _____%

ВКР выполнена с оценкой _____

Дата защиты «_____» _____ 20___ г.

Секретарь ГЭК Болдырева Е.А. _____

Листов хранения _____

Демонстрационных материалов/Чертежей хранения _____

Министерство науки и высшего образования Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

”НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО“

УТВЕРЖДАЮ

Руководитель ОП

Бессмертный И.А. _____

«_____» _____ 20___ г.

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ

Обучающийся Губарев В.Ю.

Группа P42111 **Факультет/институт/кластер** ПИиКТ

Квалификация магистр

Направление подготовки 09.04.04 Программная инженерия

Направленность (профиль) образовательной программы

Информационно-вычислительные системы

Специализация Интеллектуальные системы

Тема ВКР Разработка и реализация методов эффективного взаимодействия процессов в распределенных системах

Руководитель Косяков Михаил Сергеевич, Университет ИТМО, доцент ФПИиКТ, к.т.н.

2 Срок сдачи студентом законченной работы «25» мая 2020 г.

3 Техническое задание и исходные данные к работе

Требуется разработать и реализовать эффективные методы межпроцессного взаимодействия в пределах одного физического узла. Межпроцессное взаимодействие как с локальными, так и с удаленными процессами должно осуществляться через единый программный интерфейс. Интерфейс должен автоматически выбирать наиболее эффективный метод межпроцессного взаимодействия и скрывать реализацию от пользователя.

4 Содержание выпускной работы (перечень подлежащих разработке вопросов)

а) Обзор предметной области и постановка цели работы.

б) Разработка и реализация методов эффективного взаимодействия процессов.

в) Экспериментальное исследование и обработка результатов.

5 Перечень графического материала (с указанием обязательного материала)

а) Гистограммы временной задержки на передачу данных для разработанных методов межпроцессного взаимодействия.

б) Принципиальные схемы разработанных методов межпроцессного взаимодействия.

6 Исходные материалы и пособия

а) Косяков М.С. Введение в распределенные вычисления. Учебное пособие / М.С. Косяков. – СПб: СПбГУ ИТМО, 2014. – 155 с.

б) Schmidt D.C. et al. Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects. – John Wiley & Sons, 2013. – Т. 2.

7 Дата выдачи задания «01» ноября 2018 г.

Руководитель ВКР _____

Задание принял к исполнению _____«01» ноября 2018 г.

Министерство науки и высшего образования Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

”НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО“

АННОТАЦИЯ
ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ

Обучающийся Губарев Владимир Юрьевич

Наименование темы ВКР: Разработка и реализация методов эффективного взаимодействия процессов в распределенных системах

Наименование организации, в которой выполнена ВКР Университет ИТМО

ХАРАКТЕРИСТИКА ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ

1 Цель исследования: уменьшение временной задержки на передачу данных между процессами в пределах одного физического узла путем разработки и применения методов эффективного межпроцессного взаимодействия.

2 Задачи, решаемые в ВКР:

а) рассмотреть существующие методы межпроцессного взаимодействия, доступные при взаимодействии процессов, находящихся на одном физическом узле;

б) произвести анализ и отбор методов межпроцессного взаимодействия для реализации новых методов межпроцессного взаимодействия;

в) разработать и реализовать эффективные методы межпроцессного взаимодействия;

г) экспериментально исследовать полученные новые методы межпроцессного взаимодействия.

3 Число источников, использованных при составлении обзора: 28

4 Полное число источников, использованных в работе: 40

5 В том числе источников по годам:

Отечественных			**Иностранных**		
Последние 5 лет	От 5 до 10 лет	Более 10 лет	Последние 5 лет	От 5 до 10 лет	Более 10 лет
3	0	0	10	8	19

6 Использование информационных ресурсов Internet: да, число ресурсов: 14

7 Использование современных пакетов компьютерных программ и технологий:

Пакеты компьютерных программ и технологий	**Раздел работы**
LaTeX	Весь текст работы и сопроводительные документы
C++17 (“International Standard ISO/IEC 14882:2014(E) Programming Language C++”)	Раздел 2, приложение А
LTTng	Раздел 3

8 Краткая характеристика полученных результатов

Разработано семейство новых методов межпроцессного взаимодействия в пределах одного физического узла, показавших существенно меньшую временную задержку на передачу данных, чем методы, использующие TCP.

9 Гранты, полученные при выполнении работы

НЕТ

10 Наличие публикаций и выступлений на конференциях по теме работы

а) 1 *Губарев В. Ю.* Реализация методов эффективного взаимодействия процессов в распределенных системах [Электронный ресурс] // Сборник тезисов докладов конгресса молодых ученых. Электронное издание. — Университет ИТМО, 2020. — URL: `https://kmu.itmo.ru/file/download/application/9365` (дата обращения: 23.05.2020).

б) 1 *Губарев В. Ю.* Реализация методов эффективного взаимодействия процессов в распределенных системах. — 04.2020. — IX Конгресс молодых ученых (ОНЛАЙН ФОРМАТ).

Обучающийся Губарев Владимир Юрьевич _____

Руководитель Косяков Михаил Сергеевич _____

«_____» _____ 20___ г.

SUMMARY

OF A GRADUATION THESIS

Student: Gubarev Vladimir Yurievich

Development and implementation of efficient inter-process communication methods for distributed systems

Nowadays distributed systems are widely spreaded. They are usually designed to work in various environments as a set of cooperating processes. At the same time capabilities of modern hardware allow to deploy groups of that processes within a single machine in order to achieve better performance. In this case efficient inter-process communication (IPC) methods become a crucial element of high-performance distributed systems.

The present work is focused on developing efficient IPC methods. Based on the most efficient IPC in Linux, shared memory and futex, it introduces new methods of low-latency IPC. They are transparently provided via a generic interface. The interface automatically and transparently for programmer uses TCP to communicate over network with remote processes and low-latency shared memory-based method for local processes.

Proposed methods show significantly lower latency with local processes than TCP-based without any additional difficulties for programmer.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Актуальность темы исследования. Для некоторых систем эффективное межпроцессное взаимодействие является критически важной частью их работы. Требование по минимизации времени обслуживания заявок системой может напрямую следовать из области применения системы, как в случае с системами для алгоритмической торговли на финансовых рынках. Обслуживание заявок множеством логически связанных процессов может быть существенно ускорено при размещении таких процессов на одном физическом узле. Современные процессоры с десятками вычислительных ядер могут обеспечить такую конфигурацию нужными ресурсами. Это позволяет использовать более эффективные методы межпроцессного взаимодействия, а именно методы на основе разделяемой памяти [9]. Эффективные методы межпроцессного взаимодействия могут использоваться для связи виртуальных машин или контейнеров в пределах машины-хозяина [24, 39]. Для связи программных модулей, исполняющихся в разных процессах для обеспечения отказоустойчивости за счет изоляции процессов на уровне ОС. Для высокопроизводительных вычислений, таких как анализ научных данных или прогнозирование погоды.

При разработке сложной многокомпонентной распределенной системы программисту необходимо сосредоточиться на логике и корректности работы самой системы. В то время как методы межпроцессного взаимодействия должны быть от него скрыты. Этого можно достичь, используя единый унифицированный интерфейс для межпроцессного взаимодействия. Это упрощает разработку, снимает необходимость сложного управления ресурсами для межпроцессного взаимодействия. А также позволяет автоматически использовать наиболее подходящие методы межпроцессного взаимодействия для данной пары процессов, что может повысить эффективность выполнения некоторых задач этой системой.

Таким образом, разработка и реализация методов эффективного межпроцессного взаимодействия, предоставляемых через единый интерфейс, необходима и обоснована. Посредством этого интерфейса программист использует методы межпроцессного взаимодействия на основе разделяемой памяти при взаимодействии с локальными процессами без необходимости перекомпиляции программы. Но поскольку зачастую нельзя разместить всю систему на одном, даже очень про-

изводительном, сервере используется TCP при взаимодействии с процессами на других физических узлах.

Объектом исследования являются методы межпроцессного взаимодействия.

Предметом исследования является временная задержка на передачу данных между процессами распределенной системы в пределах одного физического узла.

Цель работы – уменьшение временной задержки на передачу данных между процессами в пределах одного физического узла путем разработки и применения методов эффективного межпроцессного взаимодействия.

В настоящей работе поставлены следующие **задачи**:

- рассмотреть существующие методы межпроцессного взаимодействия, доступные при взаимодействии процессов, находящихся на одном физическом узле;
- произвести анализ и отбор методов межпроцессного взаимодействия для реализации новых методов межпроцессного взаимодействия;
- разработать и реализовать эффективные методы межпроцессного взаимодействия;
- экспериментально исследовать полученные новые методы межпроцессного взаимодействия.

Методы исследования включают в себя анализ существующих методов межпроцессного взаимодействия, экспериментальное исследование разработанных методов межпроцессного взаимодействия и методы математической статистики для обработки экспериментальных данных.

Средства исследования:

- язык программирования C++, компилятор *Clang 6.0.1*, стандартная библиотека C++ *libstdc++*;
- библиотека Boost.Interprocess [15] для управления разделяемой памятью;
- система трассировки событий [3] на основе инструмента LTTng [23].

Научная новизна заключается в предложенных новых методах эффективного межпроцессного взаимодействия в пределах одного физического узла, которые не описаны в существующих исследованиях.

Положения, выносимые на защиту. Методы межпроцессного взаимодействия:

— через очередь в разделяемой памяти с оповещением о появлении данных в очереди через мультиплексор в разделяемой памяти и обслуживанием соединений по модели ”Лидер/Последователи“ с ожиданием сигналов потоком в режиме сна на futex;
— через очередь в разделяемой памяти с оповещением о появлении данных в очереди через мультиплексор в разделяемой памяти и обслуживанием соединений по модели ”Полусинхронный/Полуреактивный“ с ожиданием сигналов потоком в режиме сна на futex;
— через очередь в разделяемой памяти с оповещением о появлении данных в очереди через мультиплексор в разделяемой памяти и обслуживанием соединений по модели ”Лидер/Последователи“ с ожиданием сигналов потоком в режиме активного опроса мультиплексора.

Апробация результатов.

Основные результаты работы были представлены на IX Конгрессе Молодых Ученых.

Результаты работы применены в платформе для торговли на финансовых рынках Tbricks от компании Itiviti.

1 ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ И ПОСТАНОВКА ЦЕЛИ РАБОТЫ

1.1 Методы межпроцессного взаимодействия

Метод межпроцессного взаимодействия – это способ осуществления взаимодействия процессов, находящихся на одном или разных физических узлах.

В Linux поддержаны следующие методы межпроцессного взаимодействия:

а) интернет-сокеты и сокеты домена Unix,
б) именованные и неименованные каналы,
в) очередь сообщений,
г) разделяемая память: SystemV и POSIX.

Рассмотрим приведенные выше методы межпроцессного взаимодействия.

1.1.1 Сокеты

Сокет – это программный интерфейс для обеспечения обмена данными между процессами. В зависимости от реализации интерфейса, позволяет взаимодействовать процессам как на разных физических узлах в составе сети, так и в пределах одного узла. Широко распространены два вида: TCP-сокеты и сокеты домена Unix. Первый работает через протокол TCP, второй – использует некоторую внутреннюю реализацию последовательной гарантированной передачи данных.

1.1.1.1 TCP-сокеты

TCP-сокет – интерфейс межпроцессного взаимодействия, использующий протокол TCP. Среди всех рассматриваемых методов только он позволяет взаимодействовать процессам на разных физических узлах в составе сети. В то же время, возможно межпроцессное взаимодействие и в пределах одного узла через механизм обратной петли, когда TCP-сообщение передается без каких-либо лишних операций в процесс-получатель, не покидая физического узла. В качестве точки соединения используется пара IP-адрес и порт.

TCP широко распространен. Сторонние системы чаще всего предоставляют именно TCP-интерфейс доступа к своим службам.

1.1.1.2 Сокеты домена Unix

Unix-сокет или IPC-сокет – интерфейс межпроцессного взаимодействия в пределах одного узла, не использующий сетевого протокола. Может работать в разных режимах: передачи потока байт, датаграмм или последовательных пакетов. Первые два соответствуют протоколам TCP и UDP. Третий – последовательный надежный канал для передачи датаграмм.

1.1.2 Каналы

1.1.2.1 Неименованные каналы

Неименованный канал – однонаправленный метод межпроцессного взаимодействия для родственных процессов. Данные, записанные в неименованный канал, остаются там до момента считывания, либо до момента завершения ссылающихся на него процессов. Размер буфера ограничен.

1.1.2.2 Именованные каналы

Именованный канал отличается от неименованного тем, что представлен в виде файла, следовательно, доступен всем процессам ОС. Кроме того, время его жизни не ограничено временем жизни использующих его процессов.

1.1.3 Очередь сообщений

Очередь сообщений – список сообщений, который хранится в пространстве ядра. Каждая очередь имеет свой уникальный идентификатор. Взаимодействие происходит путем вызовов записи и чтения сообщений.

1.1.4 Разделяемая память

Обычно в целях безопасности адресное пространство процесса изолировано от адресных пространств других процессов. В некоторых случаях, однако, может быть необходимо использовать совместно один и тот же сегмент памяти сразу несколькими процессами. Такая память называется разделяемой. На рисунке 1 приведен пример использования разделяемой памяти двумя процессами.

Существуют два интерфейса для доступа к разделяемой памяти. Более старый System V и более новый POSIX.

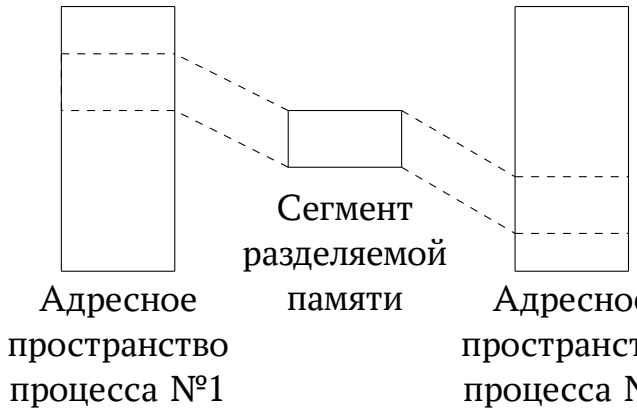

Рисунок 1 – Пример использования разделяемой памяти

1.1.4.1 System V

Разделяемая память здесь является системным ресурсом, она представлена в ОС уникальным целочисленным ключом.

Интерфейс реализуется через набор системных вызовов и структур для работы с ним:

- shmget(key_t key, size_t size, int oflag) – вызов для создания нового сегмента разделяемой памяти или использования существующего с ключом *key* . Размер сегмента *size*, флагb доступа и создания *oflag*;
- shmat(int shmid, const void *shmaddr, int flag) – подключение сегмента в адресное пространство процесса;
- shmdt(const void *shmaddr) – отключение сегмента от адресного пространства;
- shmctl(int shmid, int cmd, struct shmid_ds *buf) – управление разделяемой памятью: изменение прав доступа, удаление, запрос статистики;

1.1.4.2 POSIX

Разделяемая память здесь является пользовательским ресурсом, в системе она представлена файлом. Интерфейс реализуется через набор системных вызовов:

- shm_open(const char *name, int oflag, mode_t mode) – открывает файл для разделяемой памяти (аналог shmget). В отличие от обычного вызова open, открытый таким образом файл не синхронизируется с диском.
- shm_unlink(const char *name) – удаляет файл (аналог shmctl);
- ftruncate(int fd, off_t length) – задает размер файла;

— fstat(int fd, struct stat *statbuf) – статистика о файле (аналог shmctl);
— mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset) – отображение файла в памяти (аналог shmat);
— munmap(void *addr, size_t length) – отключает отображенный сегмент от адресного пространства процесса (аналог shmdt).

1.1.4.3 Разница между System V и POSIX

Оба интерфейса предлагают аналогичные возможности. Разница состоит в предоставлении ресурса в системе. Также, в POSIX интерфейсе сегмент разделяемой памяти будет уничтожен, когда будет завершен последний процесс, отображающий его в память, а файл полностью удален. Это свойство полезно на случай непредвиденного завершения процессов, например, в результате программного дефекта.

1.1.5 Сравнение методов межпроцессного взаимодействия

Приведенные выше методы можно разделить на группы по двум критериям:
— способности к взаимодействию с удаленными процессами, то есть с процессами на других физических узлах;
— использованию ядра ОС для осуществления межпроцессного взаимодействия.

Среди всех представленных методов только TCP-сокеты позволяют взаимодействовать как процессам на разных физических узлах, так и на одном узле, используя один и тот же интерфейс. Поэтому разумно использовать именно этот метод в качестве базового для межпроцессного взаимодействия.

Все приведенные методы, за исключением разделяемой памяти, при межпроцессном взаимодействии используют ядро операционной системы. Использование ядра существенно увеличивает временную задержку на передачу данных. Как минимум, из-за двойного копирования данных, перехода между пользовательским режимом и режимом ядра (см. рисунок 2). Разделяемая память же позволяет осуществлять взаимодействие напрямую через пользовательское пространство.

Однако, сама по себе разделяемая не предоставляет возможностей для синхронизации процессов, не имеет механизма оповещения об изменении состояния

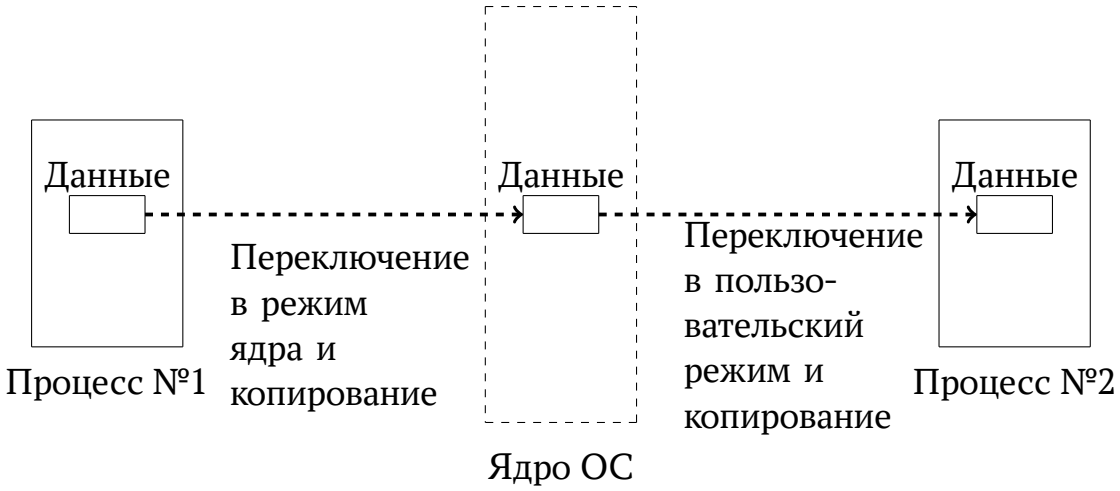

Рисунок 2 – Общая схема межпроцессного взаимодействия при использовании ядра ОС

разделяемой памяти. Для этого необходимо рассмотреть методы синхронизации процессов.

1.2 Методы синхронизации процессов

Методы синхронизации процессов нужны, чтобы корректно передавать данные между процессами через разделяемую память, например, не допускать состояния гонки. Можно выделить два класса методов: примитивы синхронизации и атомарные операции. Примитивы синхронизации зачастую используют в своих алгоритмах атомарные операции.

1.2.1 Примитивы синхронизации

В Linux на уровне ядра ОС поддержаны следующие примитивы межпроцессной синхронизации:

а) System V семафоры;
б) futex – **f**ast **u**serspace mu**tex**.

В свою очередь, futex служит основой для более продвинутых примитивов синхронизации. Среди них:

а) POSIX-семафор;
б) mutex – взаимное исключение;
в) read/write-mutex – взаимное исключение для одного писателя и множества читателей;
г) spinlock – взаимное исключение методом холостого ожидания (busy wait);

д) условная переменная – оповещение о наступлении события и ожидание события.

Рассмотрим их подробнее.

1.2.1.1 futex

Futex [13] – это низкоуровневый механизм блокировок в пользовательском пространстве. В основном, операции с ним производится с помощью атомарных инструкций в пользовательском пространстве. Ядро применяется для ожидания и диспетчеризации использующих futex процессов.

В пользовательской памяти futex – это 4-байтное число, выровненное также на 4 байта. Он может располагаться в собственном адресном пространстве процесса и использоваться для синхронизации потоков процесса, а может и в разделяемой памяти и отвечать за синхронизацию разных процессов в пределах одного физического узла. Алгоритм работы с ним устанавливается пользователем. В нужных сценариях алгоритма используется ядро ОС для разрешения состязательности, задач ожидания и диспетчеризации.

1.2.1.2 Семафоры

Семафор – это примитив синхронизации на основе целочисленного счетчика. Поддерживает две операции: увеличение значения счетчика и уменьшение. Уменьшение до нуля приводит к добавлению процесса в очередь ожидания семафора. Увеличение от нуля – к пробуждению одного из потоков из очереди ожидания.

В Linux представлены два интерфейса семафоров: System V и POSIX.

POSIX Реализованы через futex. Представляют из себя целое знаковое число. Процедуры *sem_post()* и *sem_wait()* используются для увеличения и уменьшения значения семафора на единицу соответственно. Первая также пробуждает процесс из очереди ожидания семафора, если необходимо, вторая – добавляет процесс в очередь ожидания, если значение семафора равно нулю или стало равно нулю после уменьшения значения. POSIX-семафоры представлены в двух видах: именованные и неименованные. Первые идентифицируются файлом, вторые располагаются в адресном пространстве процесса или разделяемой памяти.

System V Реализованы в ядре ОС. Идентифицируются целочисленным ключом в пределах одной ОС. Имеют возможность контролировать доступ к операциям над семафором. В отличие от POSIX-семафоров позволяет менять состояние семафора больше, чем на единицу.

1.2.1.3 Взаимные исключения

Взаимное исключение – примитив синхронизации для обеспечения эксклюзивного доступа к данным, защищенным этим примитивом. Существует несколько вариантов: обычный – mutex, рекурсивный – recursive mutex, с одновременным доступом нескольких читателей – read/write-mutex. По определению, mutex может освободить только тот поток, который его занял. Рекурсивность – возможность владельца взаимного исключения захватывать вновь.

В Linux реализованы через futex. Предоставляются посредством библиотеки *pthread*.

Mutex Классический механизм взаимного исключения. Представлен платформо-зависимым типом данных *pthread_mutex_t* Используется через семейство функций библиотеки *pthread* – *pthread_mutex_**.

RW-Mutex Версия взаимного исключения, допускающая совместное нахождение в критической секции нескольких читателей, но требующая эксклюзивного доступа для писателя. Представлена платформо-зависимым типом данных *pthread_rwlock_t* Используется через семейство функций библиотеки *pthread* – *pthread_rwlock_**.

Spinlock Реализация механизма взаимного исключения с использованием механизма холостого ожидания. Взятие блокировки состоит в изменение значения переменной на условные ”свободно“ и ”занято“, а ожидание осуществляется – в непрерывном циклическом ожидании состояния ”свободно“.

1.2.1.4 Условная переменная

Условная переменная – сигнальный примитив синхронизации. Обеспечивает блокирование одного или нескольких потоков до наступления события о выполнения предиката. Поток, меняющий параметры предиката, должен сигнализировать об этом через условную переменную, чтобы ожидающие потоки были раз-

бужены. Представлена платформо-зависимым типом данных *pthread_cond_t*. Используется через семейство функций библиотеки *pthread* – *pthread_cond_** . Условная переменная используется вместе с mutex, который обеспечивает синхронизированный доступ к параметрам предиката и другим данным.

В Linux примитив также реализован через futex.

1.2.1.5 Сравнение методов межпроцессной синхронизации

Наиболее универсальный из рассмотренных классических примитивов синхронизации – это семафор. Используя его можно реализовать как взаимное исключение, так и условную переменную. Он подходит для решения задач читателей и писателя и производителя и потребителей), как и комбинация условной переменной с mutex.

Как можно заметить, почти все рассмотренные примитивы в Linux реализованы через futex. Он может быть полезен при разработке своих собственных специализированных методов межпроцессной синхронизации [10].

1.2.2 Атомарные операции

Атомарная операция – это операция, которая выполняется исключительно целиком, либо не выполняется вовсе. В настоящей работе рассматриваются именно простейшие операции. Обычно они представлены в виде процессорных инструкций и префиксов к ним. Используются через стандартную библиотеку языка C++ как набор классов *std::atomic*.

Виды атомарных операций:

а) load/store – атомарные чтение или запись;
б) swap – атомарно устанавливает значение переменной и возвращает старое значение;
в) compare-and-swap – атомарное изменение значения переменной, если ее текущее состояние совпадает с ожидаемым;
г) fetch-and-(add/dec/or/and) – атомарная арифметическая или логическая операция над переменной с возвращением предыдущего значения.

Атомарные операции используются при разработке методов синхронизации и неблокирующих алгоритмов, структур данных с конкурентным доступом. Даже в простейшей реализации взаимного исключения, в spinlock, исполнение в критической секции требует двух атомарных операций на взятие и освобождение. Если

внутри критической секции необходимо, например, выставить бит в 4-байтной переменной, то можно сделать это, используя атомарную операцию fetch-and-or.

Вместе с futex они являются наиболее низкоуровневыми блоками для построения методов межпроцессной синхронизации.

1.3 Предыдущая работа

В ранних работах автора была разработана и реализована прикладная программная платформа для эффективного взаимодействия процессов [2] по модели ”точка-точка“. Разработан и реализован метод эффективного межпроцессного взаимодействия посредством мультиплексора в разделяемой памяти [1]. Этот метод предоставляется через реализованную ранее прикладную программную платформу, то есть не требует со стороны программиста никаких изменений в программном коде для использования разработанного метода.

1.4 Обзор литературы

1.4.1 Научные работы

Во множестве работ предлагаются методы эффективного межпроцессного взаимодействия, системы передачи сообщений между процессами в различных применениях. В подавляющем большинстве работ в качестве среды для передачи данных используется разделяемая память.

Авторы в работе [26] предлагают гетерогенную систему для передачи сообщений по модели ”издатель/подписчики“. Посредством очереди разделяемой памяти данные передаются между процессами, написанными на разных языка: Python и C++. Синхронизация и ожидание данных осуществляется через семафор.

В работе [16] представлена система для передачи сообщений, использующая разделяемую память для взаимодействия процессов в пределах одного узла и TCP – с удаленными процессами. Поддержаны взаимодействия как по модели ”точка-точка“, так и “издатель/подписчики”. Система предоставляет единый интерфейс межпроцессного взаимодействия, то есть использование того или иного метода не требует изменения исходного кода программистом.

В параллельных вычислениях широко используется MPI – Message Passing Interface. Это программный интерфейс для взаимодействия процессов посредством передачи сообщений. В ряде работ предлагаются различные методы меж-

процессного взаимодействия на основе разделяемой памяти с целью увеличения производительности параллельных систем в различных реализациях MPI, например, MPICH2 [5, 12] или MVAPICH2 [6, 38].

В работе [18] предлагается метод эффективного межпроцессного взаимодействия с использованием разделяемой памяти для встраиваемых многоядерных систем, где необходимы одновременно производительные и не ресурсоемкие методы.

Существуют работы об использовании разделяемой памяти для взаимодействия процессов в разных виртуальных машинах в пределах машины-хозяина [11, 24, 30, 39]. В том числе, работа [31], в которой разделяемая память используется как для взаимодействия процессов на одном физическом узле, так и на разных узлах в пределах кластера через удаленный прямой доступ в память (RDMA), что требует специального оборудования. В работе [40] авторы используют разделяемую память для передачи данных через сокеты (TCP и UDP) для повышения производительности взаимодействия виртуальных машин на одном физическом узле.

Эффективное межпроцессное взаимодействие также важно в роботизированных системах. Так, в работе [19] предлагается система передачи сообщений по модели ”издатель/подписчики“ посредством UDP-мультивещания с целью добиться минимальной временной задержки на передачу данных. Метод межпроцессного взаимодействия через разделяемую память предлагается в работе [36], где авторы предлагают систему передачи данных по модели ”издатель/подписчик“ с передачей управляющего блока данных через сокеты, а самих данных через разделяемую память без преобразования передаваемого объекта в поток байт.

В различных работах исследуется производительность механизмов операционных систем для межпроцессного взаимодействия. В работе [7] сравниваются системные мультиплексоры событий *select, epoll u poll*. В работе [35] проведен анализ влияния специфики работы этих механизмов на производительность методов обслуживания соединений ”Лидер/Последователи“ и “Полусинхронный-/Полуасинхронный“. А в работе [34] были предложены и исследованы модели этих методов обслуживания. В работах [9, 37] в Linux показано превосходство методов межпроцессного взаимодействия на основе разделяемой памяти над каналами, TCP и сокетами домена Unix. В работе [17] исследуются механизмы пробуждения потоков для обслуживания асинхронных событий и предлагается ме-

тод существенного уменьшения временной задержки на пробуждение потока для обслуживания асинхронного события путем использования привилегированных возможностей аппаратного обеспечения.

1.4.2 Коммерческие решения для передачи сообщений

Наиболее известные высокопроизводительные решения для передачи сообщений между процессами – это системы от компаний TIBCO [32], Informatica [20] и Solace [33]. Они представляют собой целостные решения организации обмена сообщениями между процессами распределенной системы. Для передачи данных между процессами в пределах одного физического узла используют методы на основе разделяемой памяти.

1.5 Значимость методов на основе разделяемой памяти в межпроцессном взаимодействия

Исходя из выше перечисленного, методы на основе разделяемой памяти имеют большое значение в современных системах обмена сообщениями. Использование разделяемой памяти позволяет улучшить производительность межпроцессного взаимодействия. Это следует как из ее механизма работы, так и из множества работ других исследователей, использующих разделяемую память для эффективного межпроцессного взаимодействия различных прикладных областях.

1.6 Важность единого интерфейса межпроцессного взаимодействия

Крайне важно соблюдать единый и неизменный интерфейс межпроцессного взаимодействия. Внедрение новых методов межпроцессного взаимодействия не должно приводить к изменениям в пользовательском исходном коде. Поэтому методы, разрабатываемые в настоящей работе, должны соблюдать ранее представленный интерфейс межпроцессного взаимодействия [2].

1.7 Критерий эффективности и постановка цели

В настоящей работе критерием эффективности принята временная задержка на передачу данных. Чем она меньше, тем более эффективным считается метод межпроцессного взаимодействия. Цель работы – уменьшение временной задержки на передачу данных между процессами в пределах одного физического узла

путем разработки и применения методов эффективного межпроцессного взаимодействия.

Чтобы уменьшить временную задержку на передачу данных, необходимо разработать и реализовать новые, более эффективные методы межпроцессного взаимодействия, чем представленные автором ранее. Они должны быть построены на основе очереди в разделяемой памяти и использовать более эффективные методы оповещения о появлении данных в этой разделяемой памяти, чем TCP.

Из проведенного сравнения методов межпроцессного взаимодействия и синхронизации и обзора литературы следует, что сама по себе разделяемая память является очень производительным методом межпроцессного взаимодействия. Однако, она не предоставляет механизмов синхронизации и отслеживания событий в ней. Поэтому необходимо также разработать и реализовать эффективный метод оповещения о появлении данных в разделяемой памяти.

Необходимо экспериментально исследовать разработанные методы межпроцессного взаимодействия и сравнить их с предложенным ранее методом с использованием разделяемой памяти и TCP для оповещения о наличии данных в очереди.

Новые методы должны следовать разработанному ранее интерфейсу межпроцессного взаимодействия и не требовать изменений в пользовательском коде для своего использования.

2 РАЗРАБОТКА И РЕАЛИЗАЦИЯ МЕТОДОВ ЭФФЕКТИВНОГО ВЗАИМОДЕЙСТВИЯ ПРОЦЕССОВ

2.1 Интерфейс межпроцессного взаимодействия

Существенная часть межпроцессного взаимодействия – это то, как программист его использует и как сделать это удобным процессом. Интерфейс сокетов в ОС Linux хорошо подходит для этого, так как использует механизм файловых дескрипторов, а значит:

— имеет стандартные примитивы чтения и записи: системные вызовы *read/readv* и *write/writev*;
— позволяет мультиплексировать множество соединений для одновременного отслеживания посредством системных мультиплексоров: *select/poll/epoll*.

А также позволяет взаимодействовать процессам как по сети, так и в пределах одного физического узла.

Удобный и унифицированный интерфейс может объединять под собой совершенно разные методы межпроцессного взаимодействия, скрывая реализацию. Программист зачастую не заинтересован в тонкостях методов передачи данных, ему необходимо реализовывать и совершенствовать пользовательскую логику приложения. Однако, поскольку процессы распределенной системы работают сообща над одними задачами, то ему важна и эффективность межпроцессного взаимодействия.

Необходима возможность:

— предоставлять другим процессам возможность для подключения и подключаться к другим процессам;
— принимать и передавать данные.

Для первого пункта хорошо подходит семейство шаблонов сетевого программирования Acceptor и Connector [27], которые при необходимости создают экземпляры пользовательских обработчиков соединений, реализующих интерфейс, представленный на листинге 1.

Поскольку существует необходимость полностью оградить программиста от тонкостей работы межпроцессного взаимодействия, чтобы упростить работу пользовательского кода и иметь возможность выбирать наиболее подходящий для данной ситуации метод межпроцессного взаимодействия. выбран событийный

Листинг 1 – Интерфейс пользовательского обработчика соединений на C++

```
class ISession : private Module
{
public:
    virtual int handle_message(const Message & message) = 0;
    virtual int send(const Message & message) = 0;
    virtual int handle_close() = 0;
};
```

асинхронный метод доставки сообщений. Для этого в интерфейсе пользовательского обработчика соединения предусмотрен метод *handle_message(const Message &)*. Нижележащая реализация метода межпроцессного взаимодействия вызывает данный метод для соответствующего соединения по факту получения ей очередного сообщения. Для отправки сообщений программисту предлагается использовать примитив *send(const Message &)*.

Система работает в событийной модели. Обработчик соединения является пассивной сущностью, в которую события доставляются посредством вызова метода *handle_message(const Message &)*. Во-первых, использование этой модели позволяет разделить пользовательскую логику и нижележащие методы межпроцессного взаимодействия, работающие по произвольным алгоритмам. Следовательно, это упрощает пользовательскую логику. Во-вторых, позволяет обслуживать множество соединений меньшим числом потоков, чем количество этих соединений, так как обработчики соединения нуждаются в потоке для своего выполнения только когда необходимо обработать очередное событие. Для отправки сообщений программисту предлагается использовать примитив *send(const Message &)*.

Данный интерфейс позволяет скрыть от программиста тонкости работы межпроцессного взаимодействия. Например, отслеживание готовности множества TCP-соединений к передаче или приему данных, состояние соединений. Или то, что взаимодействие происходит через разделяемую память.

2.1.1 Базовый метод межпроцессного взаимодействия на основе TCP

TCP – это наиболее широко-используемый и универсальный метод межпроцессного взаимодействия. Протокол гарантирует, что сообщения будут получены в том же количестве и порядке, в котором они были отправлены. Посредством интерфейса сокетов он позволяет взаимодействовать как с локальными, так и с

удаленными процессами распределенной системы. Более того, сторонние системы тоже зачастую предоставляют возможность именно подключения по TCP.

Кроме распространенности и удобства для программиста, он также предоставляет очень важную возможность отслеживания времени жизни соединения. Это важно, так как случае краха процесса операционная система в ходе освобождения системных ресурсов также закроет все TCP-соединения процесса и другие стороны межпроцессного взаимодействия смогут корректно обработать закрытие соединений с ним.

2.1.2 Динамическое конфигурирование соединений

Пользовательский код отвязан от реализации межпроцессного взаимодействия интерфейсом. Это позволяет произвольным образом менять конфигурацию соединения: непосредственно сам метод межпроцессного взаимодействия, добавлять необязательные шаги преобразования данных, например, поточное сжатие. Каждый шаг представлен в виде отдельного модуля, реализующего определенный интерфейс (см. листинг 2). Модули связаны в двусвязный список.

Листинг 2 – Интерфейс низкоуровневого обработчика соединений на C++

```
class Module
{
    Module * m_prev;
    Module * m_next;

public:
    virtual int handle_open() = 0;
    virtual int handle_input(size_t size, void * data) = 0;
    virtual int handle_output(size_t size, void * data) = 0;
    virtual int handle_close() = 0;
};
```

Постоянными остаются модуль работы с TCP-соединением и пользовательский модуль, промежуточные модули произвольны. Их состав определяется взаимодействующими процессами во время установления соединения. Пример приведен на рисунке 3. В нем сообщения сжимаются перед отправкой и распаковываются при получении из TCP-соединения некоторым методом поточного сжатия.

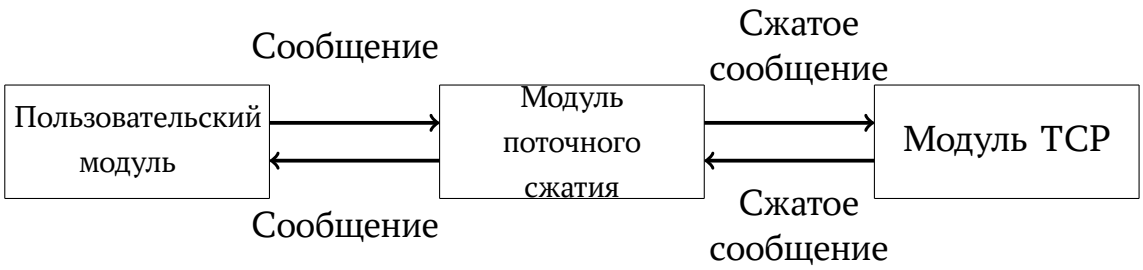

Рисунок 3 – Схема взаимодействия процессов на разных физических узлах

2.2 Применение разделяемой памяти для передачи данных

Применение разделяемой памяти при межпроцессном взаимодействии обосновано следующими причинами:

— уменьшение числа копирований данных – достаточно скопировать сообщение в очередь в разделяемой памяти и использовать эту копию в процессе-читателе при обработке;
— можно избежать системных вызовов при работе с разделяемой памятью.

Очередь при этом может быть построена произвольным образом. Например, существуют неблокирующие алгоритмы циклических буферов фиксированного размера с различными гарантиями конкурентного доступа.

В настоящей работе для простоты используется очередь на циклическом буфере постоянного размера. Она поддерживает запись данных произвольного размера в рамках размера буфера и защищается циклической блокировкой, расположенной в том же сегменте разделяемой памяти. Схема очереди представлена на рисунке 4.

Также отдельной важной проблемой является одна из проблем задачи производителя и потребителя – несоответствие скорости записи и чтения. В таком случае очередь в разделяемой памяти может быть переполнена и необходимо определить поведение системы в такой ситуации. Решение этой задачи требует отдельного исследования. Для простоты в настоящей работе предлагается разрывать соединение. Это позволит пользователю и программисту понять, что в системе дефект.

Однако, недостаточно разместить очередь в разделяемой памяти и отправлять в нее сообщения. Кроме контроля времени жизни соединения, существенный вопрос в данном методе: кто, как и когда будет считывать из очередей сообщения и обслуживать их. На одном физическом узле может находится множество взаимодействующих процессов с большим количеством соединений между ними. Со-

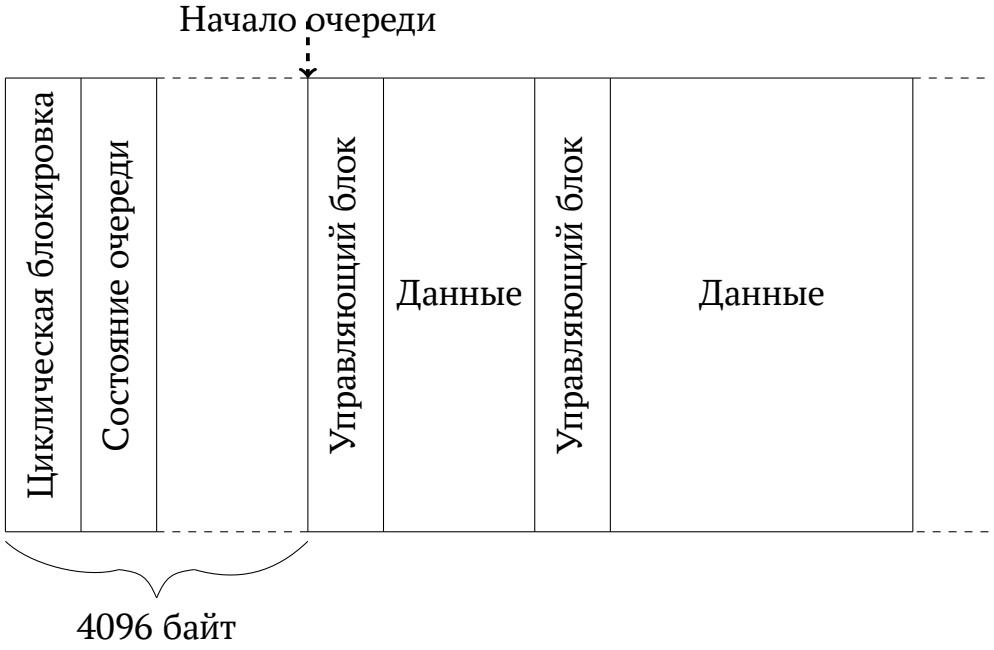

Рисунок 4 – Схема очереди в разделяемой памяти

ответственно, для каждого соединения появляется своя точка синхронизации, в которой процесс-читатель может узнать, что очередь не пуста.

Необходимо разработать метод оповещения процесса-читателя о появлении данных в разделяемой памяти, которые ему необходимо принять и обработать.

2.3 Методы оповещения о появлении данных в разделяемой памяти

2.3.1 Классические примитивы синхронизации в разделяемой памяти

Проблема ожидания готовности разделяемого ресурса хорошо известна и называется проблемой производителя и потребителя. Обычно она решается с использованием семафора или условной переменной. Алгоритмы работы в обоих случаях схожи:

- процесс-читатель ожидает готовности очереди в разделяемой памяти через примитив синхронизации;
- процесс-писатель записывает сообщение в очередь;
- процесс-писатель оповещает читателя о готовности очереди;
- процесс-читатель обрабатывает сообщение из очереди.

2.3.1.1 Применимость, достоинства и недостатки

Проблема заключается в том, что соединения становятся активными независимо друг от друга в разные моменты времени. А ожидание готовности каждого

соединения – это пассивное ожидание события на примитиве синхронизации потоком. При большом количестве соединений это может привести к созданию большого количества потоков, что может негативно сказаться на работе планировщика процессов ОС, приводит к излишней конкуренции потоков за процессорные ядра, что может ухудшать производительность системы.

Такой подход, однако, применим в системах с небольшим количеством соединений в каждом процессе, когда их число значительно меньше числа процессорных ядер.

Данный метод в настоящей работе не рассматриваются, поскольку количество соединений между процессами на одном физическом узле и самих процессов может быть большим. А значит, необходимо отслеживать оповещения сразу для множества соединений.

2.3.2 TCP

Как было сказано выше, TCP используется как базовый метод межпроцессного взаимодействия. Он может быть использован и как метод оповещения о появлении данных в очереди в разделяемой памяти.

2.3.2.1 Алгоритм взаимодействия при использовании TCP для оповещения о появлении данных

При использовании TCP для оповещения о появлении данных в очереди в разделяемой памяти алгоритм следующий:

а) процесс-читатель ожидает новых данных по всем своим TCP соединениям в состоянии сна в системном мультиплексоре оповещений;
б) для передачи данных таким методом процессу-писателю необходимо записать в очередь нужное сообщение и передать на нижележащий модуль сообщение минимального размера в 1 байт с заранее установленным значением (например, ”0“);
в) ядро операционной системы пробуждает процесс-читатель;
г) процесс-читатель демультиплексирует активное TCP соединение, считывает 1 байт и отправляет его на следующий слой обработки межпроцессных взаимодействий через ранее описанный стек модулей (см. рисунок 5);
д) модуль, отвечающий за взаимодействие по разделяемой памяти, проверяет, что полученный 1 байт имеет заранее оговоренное значение (”0“ в приме-

ре выше) и это служит для него сигналом к проверке состояния очереди в разделяемой памяти для соединения, с которого этот сигнальный байт был получен;

е) процесс-читатель считывает сообщение из очереди и выполняет его обслуживание.

Таким образом происходит передача данных между процессами в разделяемой памяти с оповещением о появлении данных в разделяемой памяти по TCP. Схема взаимодействия представлена на рисунке 5.

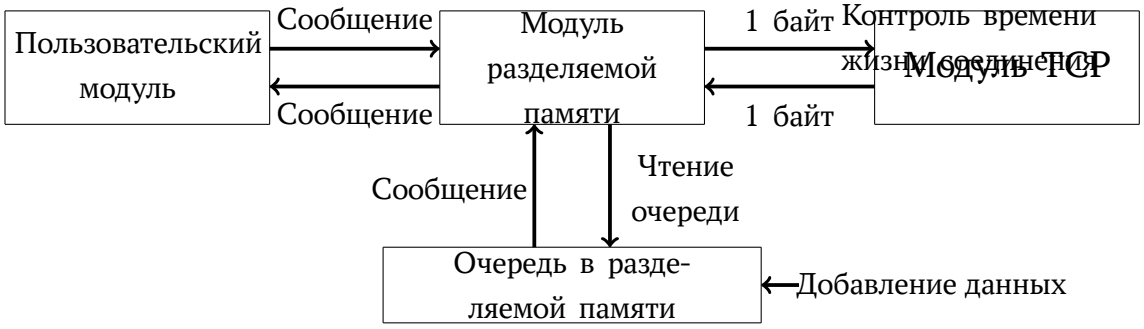

Рисунок 5 – Схема взаимодействия процессов с использованием разделяемой памяти и TCP

2.3.2.2 Достоинства и недостатки

Описанный выше метод обладает следующими **достоинствами**:

— позволяет эффективно поддерживать множество соединений с использованием системных мультиплексоров оповещений (*select/poll/epoll*);

— позволяет процессу-читателю блокировать свое выполнение до появления оповещения;

— С использованием повторной проверки состояния очереди (см. подраздел 2.2) временная задержка на передачу данных может быть снижена, если к концу обработки очередного сообщения в очередь уже будет записано новое сообщение;

— метод межпроцессного взаимодействия по TCP не требует доработки и может быть использован как есть.

Недостаток у такого подхода только один: временная задержка на отправку и получение оповещения. Основные отличия от метода, использующего только TCP, в том, как передается само сообщения. Но кроме использования разделяемой памяти для передачи самих данных выполняется также ряд системных вызо-

вов для оповещения процесса-читателя, каждый из которых может вносить существенную временную задержку:

— *write* – для записи 1 байта в TCP-сокет процессом-писателем;
— *select/poll/epoll* – для поиска активного соединения среди всего их множества процессом-читателем;
— *read* – для чтения 1 байта из TCP-сокета процессом-читателем.

В случае, когда процесс-читатель находится в состоянии сна на системном мультиплексоре оповещений, процесс-писатель в ходе системного вызова *write* должен также изменить состояние процесса-читателя на ”Готов к выполнению“ или ”Выполняется“. После этого в течение некоторого промежутка времени процесс-читатель будет готовиться к выполнению. Все это может влиять на временную задержку на передачу данных.

Следовательно, необходимо разработать новый метод мультиплексирования соединений, использующих разделяемую память, имеющий меньшие накладные расходы на использование и обладающий, как минимум, теми же достоинствами, что и описанный в данном пункте.

2.3.3 Мультиплексор оповещений в разделяемой памяти

С целью избежать излишних накладных расходов и использовать системные ресурсы наилучшим образом в настоящей работе предлагается метод мультиплексирования оповещений от множества соединений, использующих разделяемую память для передачи данных.

Мультиплексор оповещений в разделяемой памяти – это структура данных, используемая множеством процессов-писателей для оповещения процесса-читателя о появлении данных в очередях в разделяемой памяти. Каждый процесс, участвующий в межпроцессном взаимодействии, должен иметь свой мультиплексор оповещений, через который ему будут поступать оповещения о появлении данных в разделяемой памяти, которые он может считать и обработать.

Время жизни мультиплексора оповещений определяется процессом-читателем. При необходимости процесс-читатель создает файл определенного размера и отображает его в свою память. Во время установления соединения процесс-читатель ассоциирует это соединение с номером от 0 до 2047 и отправляет его вместе с путем до файла другой стороне. В будущем эти данные будут использованы противоположной стороной для отправки оповещений.

Схема взаимодействия при использовании мультиплексора оповещений представлена на рисунке 6.

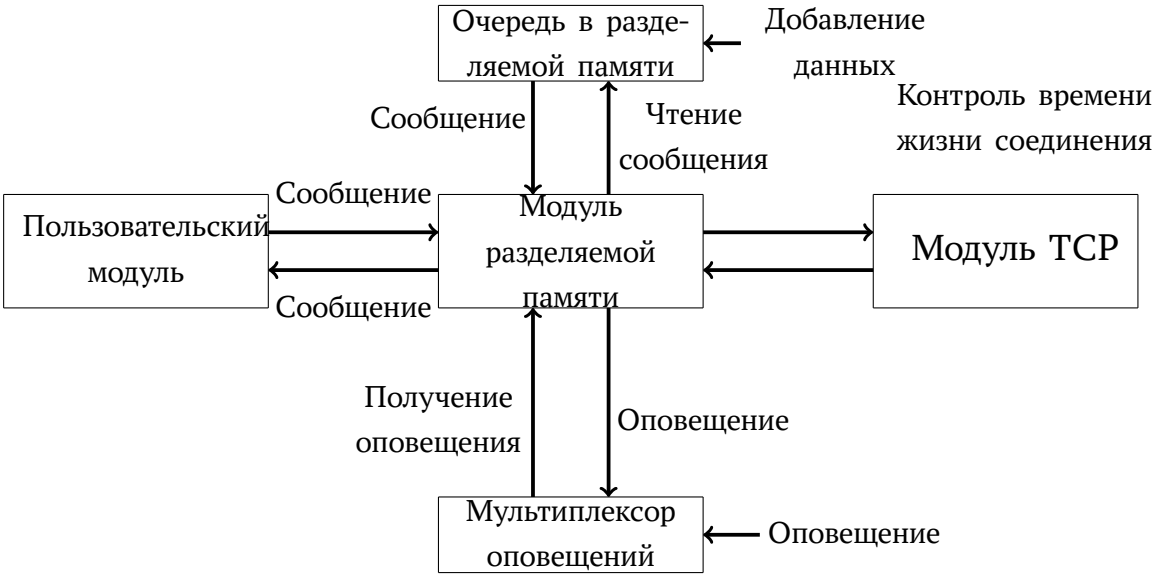

Рисунок 6 – Схема взаимодействия процессов с использованием разделяемой памяти и мультиплексора оповещений

2.3.3.1 Структура и алгоритм работы мультиплексора оповещений в разделяемой памяти

Структура мультиплексора оповещений представлена на рисунке 7. Он состоит из 4-байтного целого числа futex, используемого для синхронизации взаимодействующих процессов, и массив из 32 8-байтных сигнальных чисел, по одному на каждый бит futex. Эти 32 8-байтных числа содержат 2048 бит, что позволяет различать 2048 различных соединений. Описание на языке C++ представлено на листинге 3.

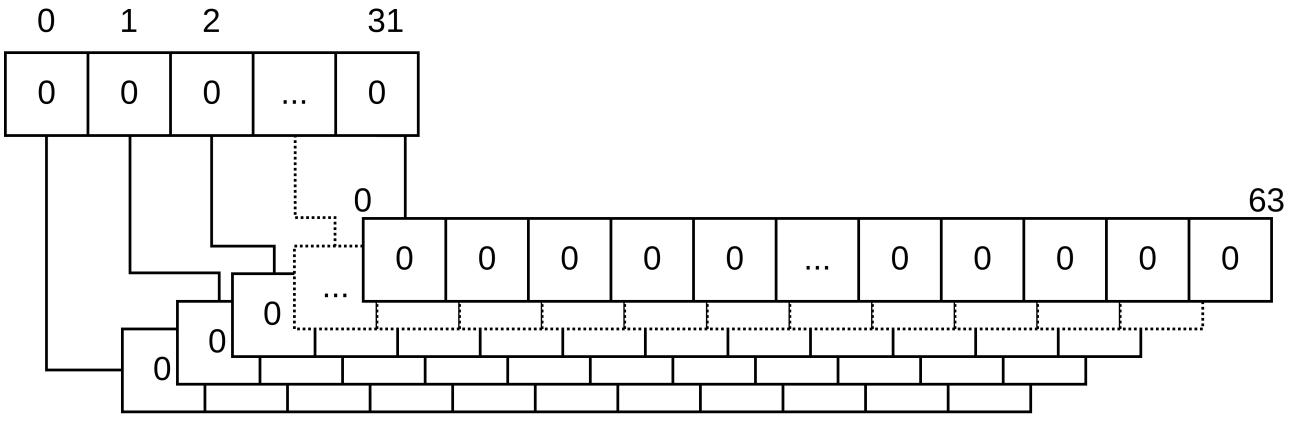

Рисунок 7 – Структура мультиплексора оповещений в разделяемой памяти

Листинг 3 – Структура мультиплексора в памяти

```
struct Multiplexer
{
    using Futex = std::atomic<int32_t>;
    using Signal = uint64_t;

    static constexpr std::size_t c_num_chunks = sizeof(Futex) *
   CHAR_BIT;
    static constexpr std::size_t c_signals_per_chunk = sizeof(std::
   atomic<Signal>) * CHAR_BIT;

    // Процедура ожидания на futex
    void wait() {
        if (!m_futex) {
            futex_wait(&m_futex, 0);
        }
    }

    // Процедура оповещения процесса–читателя. Выставляет
   соответствующие биты мультиплексора для сигнала за номером signal
   и при необходимости пробуждает поток мультиплексора процесса–
   читателя.
    void notify(Multiplexer::Signal id);

    // Процедура для пробуждения потока, спящего на futex.
    void wakeup();

protected:
    // –байтное4 число futex, на котором происходит синхронизация
   снапробуждения/ потока мультиплексора процесса–читателя.
    Futex m_futex;

    // Для избежания лишней состязательности между атомарными
   операциями над массивом сигнальных чисел и над futex, массив
   выровнен на размер кэшлинии- процессора – 64 байта.
    alignas(64) std::array<std::atomic<Signal>, c_num_chunks>
   m_signals;
};
```

Когда процесс-писатель хочет оповестить процесс-читатель о появлении данных в очереди в разделяемой памяти, ему необходимо:

а) атомарно выставить бит в сигнальном числе, соответствующий этому соединению;

б) атомарно выставить соответствующий бит futex и получить его предыдущее значение;

в) Если предыдущее значение равно нулю, значит, разбудить процесс-читатель, чтобы он мог обработать оповещение.

г) Если оно не равно нулю, это значит, что другой процесс-писатель уже либо разбудил целевой процесс-читатель, либо в скором времени сделает это (так как он был тем самым процессом, перед которым в futex был нуль).

После завершения первых двух этапов мультиплексор для соединения под номером #1987 будет находиться в состоянии, представленном на рисунке 8. Чтобы проставить нужные биты для данного соединения, процесс-писатель делит номер его соединения на 64 и получает индекс бита в futex – 31, и, соответственно, индекс сигнального числа. Далее он вычисляет остаток от деления номера сигнала на 64 и получает индекс бита в сигнальном числе – 3. После чего атомарной операцией ”ИЛИ“ выставляется сначала нужный бит в сигнальном числе, а потом нужный бит в futex. Псевдокод алгоритмов процесса-писателя и читателя приведены на листингах 4 и А.1.

Листинг 4 – Исходный код процедуры оповещения процесса через мультиплексор оповещений в разделяемой памяти

```
void Multiplexer::notify(Signal id) {
    // Шаг 1. Выставить нужный бит в одном из сигнальных чисел в
   массиве. Число находится как результат деления номера соединения
   на 64, те.. число от 0 до 31, позиция бита как остаток от деления
   номера соединения на 64.
    m_signal[id / Multiplexer::c_signals_per_chunk].fetch_or(1 << id
   % Multiplexer::c_signals_per_chunk);
    // Шаг 2. Выставить бит futex, соответствующий сигнальному
   числу. Позиция нужного бита находится как результат деления
   нмоера соединения на 64, те.. число от 0 до 31.
    uint32_t futex = m_futex.fetch_or(1 << id / Multiplexer::
   c_signals_per_chunk);
    if (!futex) {
        // Шаг 3. Если предыдущее значение futex было 0, то
   попытаться разбудить один процесс, спящий на futex, системным
   вызовом futex.
        this->wakeup();
    }
}
```

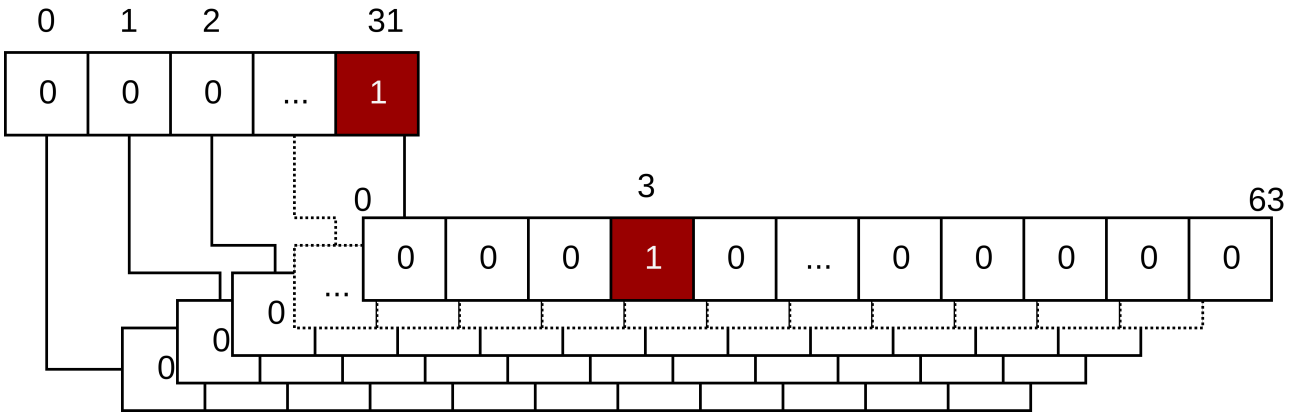

Рисунок 8 – Состояние мультиплексора оповещений в разделяемой памяти с активным сигналом #1987

2.3.3.2 Достоинства и недостатки

Данный метод решает проблему, описанную в разделе 2.3.1, позволяя определять соединение-источник сигнала за время, не зависящее от количества соединений.

По сравнению с вышеописанным методом межпроцессного взаимодействия, использующим TCP для оповещения о появлении данных в разделяемой памяти, данное решение обладает следующими **достоинствами**:

а) Подавляющая часть работы происходит в пользовательском пространстве. В лучшем сценарии отправка и получение оповещения не задействуют системные вызовы. Оповещение происходит посредством двух атомарных операций.

б) Ядро ОС используется только пробуждения и ожидания оповещений в режиме сна.

Таким образом, в худшем случае совершается один системный вызов для пассивного ожидания оповещений в режиме сна и один для пробуждения процесса. Пробуждение потока влечет временные затраты как со стороны процесса-писателя – на пробуждение потока, так и со стороны процесса-читателя – на уход в состояние сна и временную задержку на постановку потока мультиплексора на выполнение. В то же время, ожидание оповещений в состоянии сна позволяет экономить ресурс процессора.

Недостатком данного метода является механизм управление файлом мультиплексора оповещений. Поскольку файлом управляет процесс-читатель, то в случае его некорректного завершения файл не будет удален и останется навсегда.

Некорректное завершение может произойти по следующим причинам: крах процесса вследствие программного дефекта, при отправке сигнала *SIGKILL* процессу, который в большинстве случаев приводит к немедленному завершению процесса.

Данный недостаток можно решить, используя более продвинутый механизм создания файлов для мультиплексора оповещений. Например, делегировать эту работу отдельному процессу или группе процессов. В таком случае такой процесс может отслеживать состояние своих клиентских процессов и при прекращении их работы выполнять очистку ресурсов.

2.3.4 Методы обслуживания соединений

Имея более совершенный механизм оповещения процессов о появлении данных в разделяемой памяти необходимо разработать также и метод обслуживания получаемых оповещений. В данном пункте предложены четыре метода:

а) Синхронный – существует единственный выделенный поток мультиплексора. Он занимается также как ожиданием и получением оповещений, так и обслуживанием самих соединений..

б) ”Полусинхронный/Полуреактивный“ – существует единственный выделенный поток мультиплексора. Он диспетчеризует обслуживание соединений по полученным в пул потоков [28].

в) ”Лидер/Последователи“ – в один момент времени только один поток-лидер пассивно отслеживает состояние мультиплексора в режиме сна. При обнаружении оповещений он передает лидерство произвольному потоку из пула и переходит к обслуживанию соединений по полученным оповещениям [22].

г) ”Лидер/Последователи“ – в один момент времени только один поток-лидер активно отслеживает состояние мультиплексора. То есть, постоянно опрашивает мультиплексор. При обнаружении оповещений он передает лидерство произвольному потоку из пула и переходит к обслуживанию соединений по полученным оповещениям.

2.3.4.1 Синхронный метод обслуживания соединений

В данном методе для обслуживания оповещений выделяется один поток, который обслуживает только соединения, использующие мультиплексор оповеще-

ний. Диаграмма его работы представлена на рисунке 9. В отсутствие заявок поток мультиплексора находится в состоянии сна. Когда необходимо обработать соединение, ядро пробуждает этот поток и он, в свою очередь, выполняет алгоритм на листинге А.1, синхронно обрабатывая соединения, для которых были получены оповещения. Алгоритм выполняется до тех пор, пока есть оповещения для обслуживания.

Метод может быть полезен для систем с малым количеством соединений. Из-за своей простоты он может показать лучшую временную задержку на передачу данных, так как нет необходимости, как в других методах, диспетчеризовать обслуживание соединения в пуле потоков или выбирать новый поток-лидер перед обслуживанием данного соединения.

Однако, поскольку поток, обслуживающий мультиплексор, один, то когда он обслуживает текущее соединение, все остальные активные соединения простаивают. Данный метод далее в работе не рассматривается, так как несмотря на потенциально более низкую временную задержку на передачу данных при его использовании, он ограничен в количестве одновременно активных соединений.

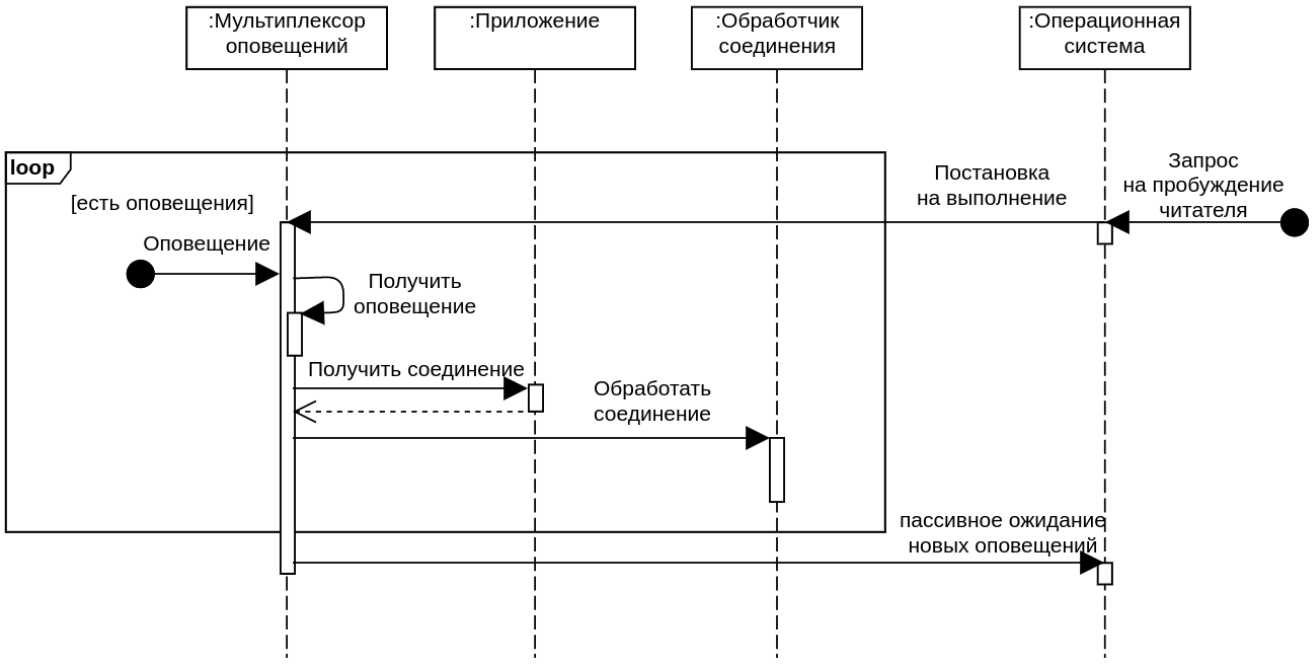

Рисунок 9 – Диаграмма синхронного обслуживания соединений выделенным потоком мультиплексора

2.3.4.2 Метод обслуживания соединений ”Полусинхронный/Полуреактивный“

В данном методе соединения по полученным оповещениям обслуживает пул потоков. Он получает задачи на обслуживание от выделенного потока мультиплексора оповещений. Применение шаблона проектирования сетевых приложений ”Полусинхронный/Полуреактивный“ [28] к синхронному методу позволяет избежать простаивания активных соединений и обслуживать параллельно столько же соединений, сколько потоков выделено в пуле. Диаграмма работы метода представлена на рисунке 10.

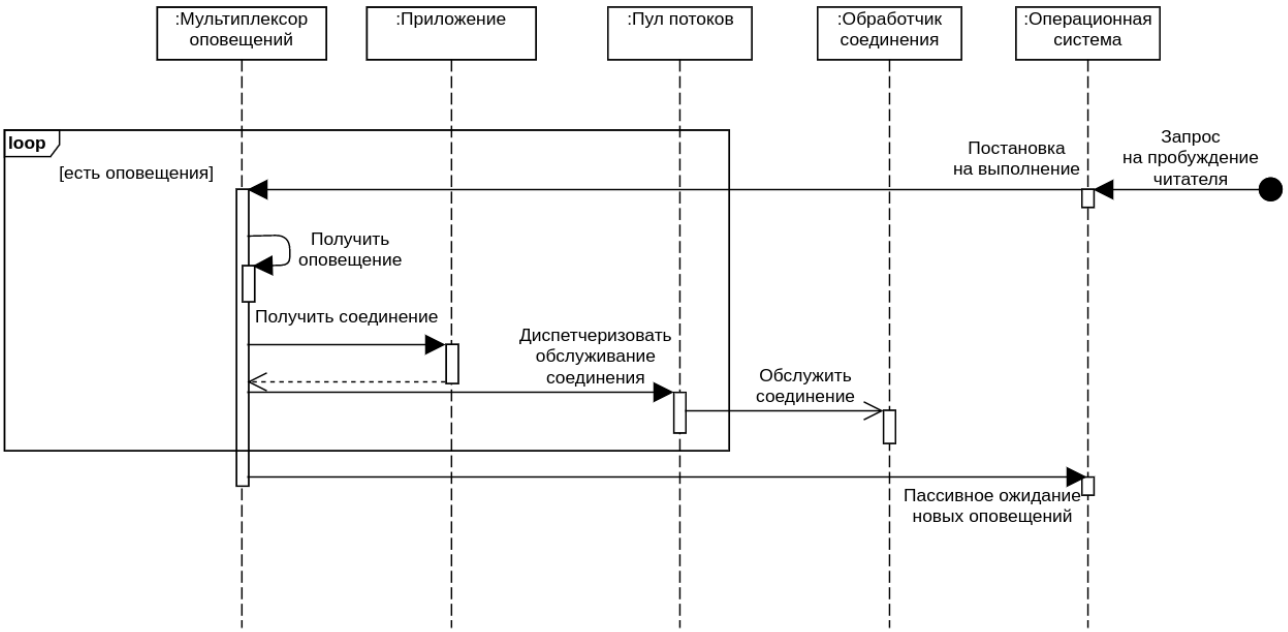

Рисунок 10 – Диаграмма диспетчеризации и обслуживания соединений по методу ”Полусинхронный/Полуреактивный“

Получение оповещений для соединения и обслуживание заявок для этого соединения происходит в разных потоках. Необходимо сохранить гарантию последовательного обслуживания заявок в соединении. То есть, в один момент времени заявки одного соединения обслуживает не более одного потока. Может произойти ситуация, при которой очередное оповещение для очередной заявки будет получено во время обслуживания текущей. В таком случае, поток из пула должен обслужить новую заявку после текущей, либо обслуживание должно быть делегировано пулу потоков после обслуживания текущей заявки. Первая стратегия предпочтительнее, так как имеет лучшую локальность по памяти и времени: один и тот же поток последовательно обслуживает заявки для одного и того же

соединения при их наличии. В обоих случаях необходимо синхронизировать диспетчеризацию и непосредственно процедуру обслуживания соединения.

На листингах 5 и 6 представлен исходный код процедур диспетчеризации и обслуживания соединений. Между собой процедуры синхронизированы механизмом взаимного исключения и набором состояний.

Листинг 5 – Процедура диспетчеризации обслуживания соединения

```
void IMultiplexerListener::check_close_n_handle_send()
{
    std::unique_lock<std::mutex> lock(m_input_close_mutex);
    if (m_state == State::Closed) {
        return;
    }
    if (m_state == State::Handling) {
        m_state = State::KeepHandling;
        return;
    }
    m_state = State::Handling;
    lock.unlock();

    m_thread_pool->dispatch_async([this] { this->process_send() });
}
```

Листинг 6 – Процедура обслуживания соединения в мультиплексоре оповещений

```
void IMultiplexerListener::process_send()
{
    while (true) {
        this->handle_send();

        std::unique_lock<std::mutex> lock(m_input_close_mutex);

        if (m_state == State::Closed) {
            lock.unlock();
            this->on_deferred_close();
            break;
        }
        else if (m_state == State::Handling) {
            m_state = State::Idle;
            break;
        }
        else { //m_state == State::KeepHandling
            m_state = State::Handling;
        }
    }
}
```

Процедура *check_close_n_handle_send()* вызывается потоком мультиплексора для диспетчеризации обслуживания соединения. В зависимости от состояния соединения, выполняется один из трех сценариев.

а) если соединение закрыто, то игнорировать оповещение из мультиплексора. Очисткой ресурсов соединения занимается либо поток, закрывающий соединение, либо поток, обслуживающий соединение;
б) если соединение уже сейчас обслуживается (или же обслуживание диспетчеризовано), то необходимо через состояние соединения заставить поток из пула произвести обслуживание еще раз;
в) если же соединение в данный момент не обслуживается, то пометить его таковым и диспетчеризовать обслуживание в пуле потоков.

Процедура обслуживания заявок в соединении *process_send()* выполняется потоком из пула. Перед завершением обслуживания соединения необходимо проверить, было ли получено новое оповещение (состояние соединения *KeepHandling*). И если да, то повторить обслуживание. Состояние *KeepHandling* необходимо, так как иначе может произойти утеря оповещения. А именно в ситуации, когда поток мультиплексора при получении оповещения видит, что соединение еще обслуживается (состояние *Handling*), а поток из пула уже закончил фактическое обслуживание и вскоре поменяет состояние соединения на *Idle*.

Закрытие соединения также необходимо синхронизировать с его обслуживанием. Если соединение обслуживается прямо сейчас, нельзя освобождать занятые им ресурсы. Необходимо дождаться завершения обслуживания и только потом провести подчистку. В настоящей работе предлагается в такой ситуации выполнять закрытие потоком, который в данный момент обслуживание соединение. Процедура *try_deactivate_listener()* проверяет состояние соединения: закрывать соединение и освобождать ресурсы можно только в состоянии *Idle*, в противном случае соединение помечается как закрытое, чтобы поток из пула мог завершить процедуру закрытия после завершения обслуживания соединения. Исходный код приведен на листинге 7.

Данный метод позволяет улучшить качество обслуживания соединений, используя параллелизм обслуживания на уровне соединений. Но использование дополнительных потоков при обслуживании может негативно влиять на временную задержку на передачу данных. Диспетчеризация и постановка потока из пула на выполнение для обслуживания заявки может занимать существенное время. Так-

Листинг 7 – Процедура закрытия соединения в мультиплексоре оповещений

```
bool IMultiplexerListener::try_deactivate_listener()
{
    std::unique_lock<std::mutex> lock(m_input_close_mutex);
    if (m_state != State::Idle) {
        lock.unlock();
        m_mux->deregister_listener(this);
        return true;
    }
    m_state = State::Closed;
    return false;
}
```

же, поток мультиплексора может ожидать новых оповещений в состоянии сна в случае их долгого отсутствия. В таком случае временная задержка на передачу данных также будет увеличена на время постановки потока мультиплексора на выполнение для получения оповещений.

2.3.4.3 Метод обслуживания соединений ”Лидер/Последователи“

Как было сказано выше, в методе обслуживания соединений ”Полусинхронный/Полуреактивный“ заявки в соединении обслуживаются потоками из пула потоков. От приема оповещения о появлении данных в разделяемой памяти до обслуживания соединения происходит диспетчеризация и постановка на выполнение потока из пула для обслуживания заявки. Использование метода ”Лидер/Последователи“ [22] позволяет избежать второго слагаемого и, возможно, получить меньшую временную задержку на передачу данных.

В рассматриваемом методе, в отличие от предыдущего ”Полусинхронный/Полуреактивый“, отсутствует выделенный поток мультиплексора. Мультиплексированием и обслуживанием занимаются потоки из пула потоков. Это позволяет избежать накладных расходов на поддержание лишнего потока, синхронизацию между потоком мультиплексора и потоками пула, временной задержки на постановку на выполнение потока для обслуживания соединений [29, с. 41]. Диаграмма работы метода представлена на рисунке 11.

В каждый момент времени состояние мультиплексора опрашивает не более одного потока, потока-лидера. При наличии оповещений для обслуживания, поток-лидер диспетчеризует в пул потоков постановку на выполнение нового лидера. Сам же он перестает быть лидером и занимается обслуживанием соеди-

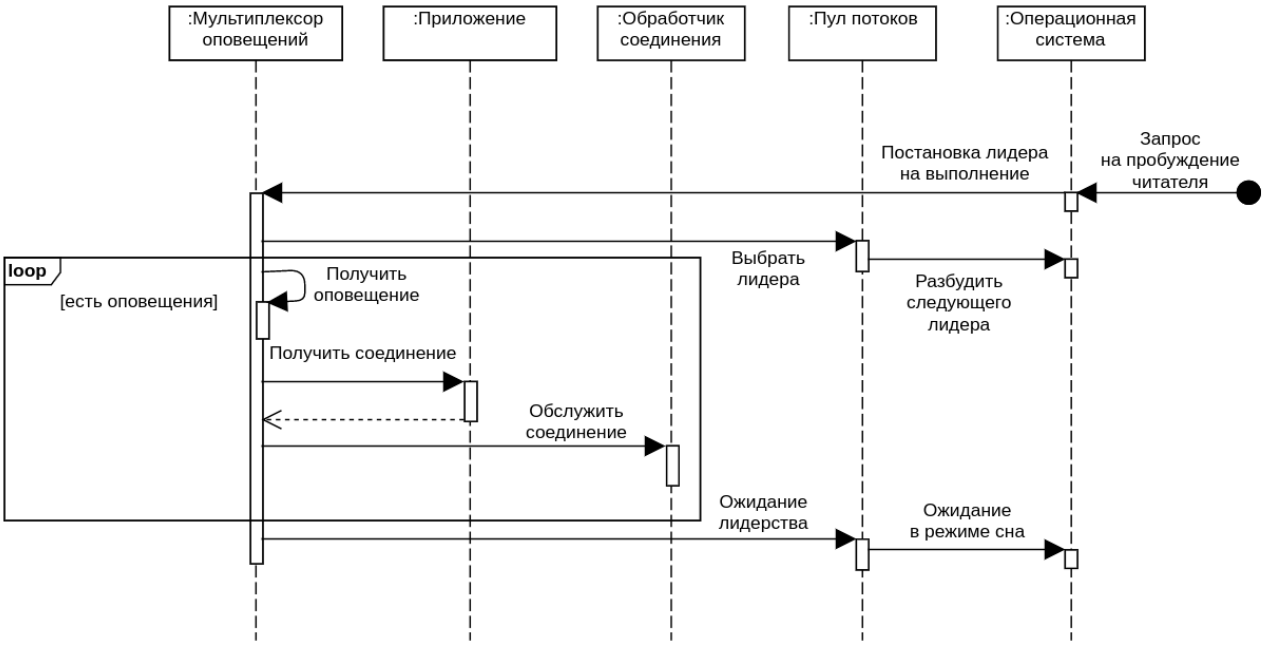

Рисунок 11 – Диаграмма обслуживания соединений по методу ”Лидер/Последователи“

нений, для которых получены оповещения. Это требует внесения изменений в процедуру опроса мультиплексора и обслуживания соединений. Изменения представлены на листинге А.2. Среди них:

а) Поток-лидер должен запомнить все соединения, которые ему необходимо обслужить.

б) Каждое из этих соединений необходимо пометить как *Handling*. Метод *should_process()* приведен на листинге 8.

в) Поток-лидер диспетчеризует нового лидера.

г) После обслуживания соединений для полученных оповещений поток переходит в состояние ожидания лидерства.

Второй шаг необходим, так как в противном случае новый лидер может получить новое оповещение для уже обслуживаемых предыдущим лидером соединений и тогда два потока будут параллельно обслуживать одно и то же соединение, что недопустимо.

В настоящей работе реализована вариация метода, в которой поток-лидер обслуживает соединения для всех обнаруженных оповещений. Такой подход может быть эффективен, когда число одновременно активных соединений мало. Недостаток такой реализации состоит в худшем использовании параллелизма на уровне соединений и накоплении временной задержки на передачу данных в слу-

Листинг 8 – Процедура подготовления соединения к обслуживанию в модели обслуживания соединений ”Лидер/Последователи“

```
bool IMultiplexerListener::should_process()
{
    std::unique_lock<std::mutex> lock(m_input_close_mutex);
    if (m_state == State::Closed) {
        return false;
    }
    if (m_state == State::Handling) {
        m_state = State::KeepHandling;
        return false;
    }
    m_state = State::Handling;
    return true;
}
```

чае, если несколько соединений последовательно будут обслужены одним потоком. Как и в предыдущем методе, поток-лидер может ожидать новых оповещений в состоянии сна на futex в случае их долгого отсутствия. В таком случае временная задержка на передачу данных также будет увеличена на время постановки потока мультиплексора на выполнение.

2.3.4.4 Метод обслуживания соединений ”Лидер/Последователи“ с активным опросом мультиплексора

Как было сказано выше, время постановки на выполнение потока, опрашивающего мультиплексор, может влиять на временную задержку на передачу данных. Потому что для передачи данных необходимо сначала доставить процессу-читателю оповещение о том, что данные для него готовы.

Мультиплексор оповещений в разделяемой памяти состоит из двух уровней (см. рисунок 7):

— 4-байтного числа futex, используемого для ожидания в режиме сна и пробуждения потока мультиплексора;
— 32 8-байтных сигнальных чисел по одному на каждый бит числа на первом уровне.

Можно использовать иной метод ожидания оповещений – активный опрос первого уровня мультиплексора вместо пребывания в режиме сна на futex. При таком подходе ядро ОС не задействуется в ходе межпроцессного взаимодействия. Схема работы метода представлена на рисунке 12.

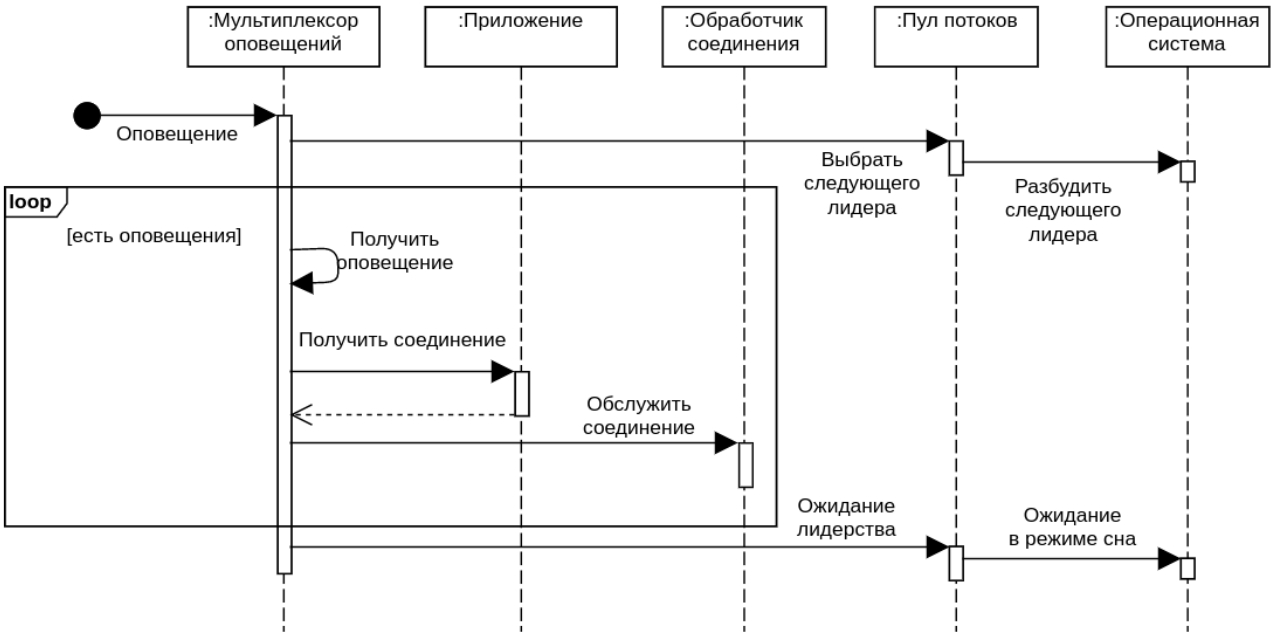

Рисунок 12 – Диаграмма обслуживания соединений по методу ”Лидер/Последователи“ с активным опросом мультиплексора оповещений

Поток, опрашивающий мультиплексор, делает это без перерыва в рамках выделенного ему планировщиком ОС кванта процессорного времени. Такой подход называется холостым ожиданием (или же busy wait). Он используется в различных реализации циклических блокировок. Метод активного ожидания в применении к взаимным исключениям критикуется за пустую трату процессорного времени [4, 14]. Но в применении к данной работе это может быть допустимо, так как позволяет уменьшить временную задержку на передачу данных при соблюдении нескольких условий, связанных с планированием процессов в ОС:

— достаточность аппаратных ресурсов. Каждый процесс, который будет использовать метод активного ожидания на мультиплексоре, будет постоянно занимать одно ядро процессора. В случае недостатка вычислительных мощностей планировщик задач может уменьшать доступное процессам процессорное время для сохранения качества обслуживания;
— время ожидания оповещений мало. В ростом времени активного опроса мультиплексора растет вероятность вытеснения потока мультиплексора с процессора. Поскольку поток мультиплексора ожидает оповещений только в пользовательском пространстве, ядру неизвестно, что поток надо будет поставить на выполнение при изменении состояния мультиплексора. Значит, временная задержка на передачу данных может значительно вырасти.

В методе обслуживания соединений ”Лидер/Последователи“ поток, который активно опрашивает мультиплексор, меняется после получения оповещений. А значит, для потока-лидера уменьшаются шансы быть вытесненным с процессора из-за окончания выделенного ему кванта процессорного времени. В отличие от рассматриваемого метода, в методе ”Полусинхронный/Полуреактивный“ только один выделенный поток занимался бы активным опросом мультиплексора, что приводило бы к его вытеснению с процессора из-за израсходования процессорного времени. Поэтому метод активного опроса мультиплексора рассматривается только для метода обслуживания соединений ”Лидер/Последователи“.

Для реализации рассматриваемого метода необходимо изменить алгоритм работы двух процедур мультиплексора оповещений: процедуры оповещения и ожидания оповещений.

На листинге 9 приведена процедура ожидания оповещений. В отличие от методов с пассивным ожиданием оповещений, в рассматриваемом методе не используется системный вызов *futex_wait*. Поток-лидер циклически опрашивает первый уровень мультиплексора до тех пор, пока не будет получено оповещение.

Листинг 9 – Исходный код процедуры ожидания оповещений через мультиплексор оповещений в разделяемой памяти при использовании метода активного опроса мультиплексора

```
void Multiplexer::wait() {
    while (!m_futex) {
        _mm_pause();
    }
}
```

На листинге 10 приведена процедура ожидания оповещений. В отличие от методов с пассивным ожиданием оповещений, в рассматриваемом методе оповещение состоит исключительно в выставлении нужных битов в мультиплексоре. Системный вызов *futex_wake* также не нужен, так как процесс-читатель не находится в состоянии сна в ожидании оповещений.

Таким образом, удается избежать системных вызовов при передаче данных между процессами и временной задержки на постановку потока-лидера на выполнение. Чтобы обслужить соединения по полученным оповещениям, необходимо диспетчеризовать новый поток-лидер для обеспечения параллелизма на уровне соединений. Метод также обладает недостатками из-за метода холостого ожида-

Листинг 10 – Исходный код процедуры оповещения процесса через мультиплексор оповещений в разделяемой памяти при использовании метода активного опроса мультиплексора

```
void Multiplexer::notify(Signal id) {
    m_signal[id / Multiplexer::c_signals_per_chunk]
        .fetch_or(1 << id % Multiplexer::c_signals_per_chunk);
    uint32_t futex = m_futex
        .fetch_or(1 << id / Multiplexer::c_signals_per_chunk);
}
```

ния: непредсказуемыми задержками из-за планировщика процессов и неэффективным использованием процессорного времени.

Выводы по главе 2

Разработан новый метод оповещения о появлении данных в очереди в разделяемой памяти посредством мультиплексора оповещений в разделяемой памяти. Мультиплексор оповещений позволяет отслеживать оповещения одновременно от 2048 соединений. Подавляющая часть работы с ним производится в пользовательском пространстве. Ядро ОС используется только для ожидания оповещений в режиме сна и для пробуждения процесса-читателя при оповещении. На его основе разработано семейство методов межпроцессного взаимодействия:

а) с пассивным ожиданием оповещений в мультиплексоре в режиме сна и обслуживанием соединений по методу ”Полусинхронный/Полуреактивный“;
б) с пассивным ожиданием оповещений в мультиплексоре в режиме сна и обслуживанием соединений по методу ”Лидер/Последователи“;
в) с активным опросом мультиплексора и обслуживанием соединений по методу ”Лидер/Последователи“.

Рассмотрены их сильные и слабые стороны, даны рекомендации по их практическому применению. Сделаны предположения об их устройстве на влияние на временную задержку на передачу данных.

3 ЭКСПЕРИМЕНТАЛЬНОЕ ИССЛЕДОВАНИЕ И ОБРАБОТКА РЕЗУЛЬТАТОВ

3.1 Постановка эксперимента

3.1.1 Конфигурация экспериментального стенда

3.1.1.1 Аппаратное обеспечение

Процессоры 2 x Intel Xeon Platinum 8168 с базовой частотой 2.7 GHz. Технология Hyper-Threading отключена для уменьшения нежелательного влияния на время обслуживания заявок в системе [21]. Счетчик тактов процессора работает в постоянном режиме на постоянной частоте 2.7 ГГц, то есть разрешение таймера $\frac{1}{2.7*10^9 \text{ГГц } 0.37}$. Это обеспечивается поддержкой так называемого Invariant TSC [8, с. 153] в используемом процессоре. Синхронизацию счетчиков тактов между процессорами обеспечивает ядро ОС.

Оперативная память: DDR4-2666 128 GiB.

3.1.1.2 Программное обеспечение

Операционная система Red Hat Enterprise Linux Server release 7.8 (Maipo). Ядро Linux 3.10.0-1127.el7.x86_64.

Компилятор C++ Clang 6.0.1.

Стандартная библиотека C++: libstdc++ 8.1.0.

Библиотека Boost.Interprocess 1.68.0.

LTTng 2.10.3

3.1.2 Конфигурация экспериментальной системы

Система для проведения эксперимента состоит из двух процессов:

— Процесс-шлюз отвечает за преобразование заявок из формата внешнего мира во внутренний формат системы и обратно.

— Процесс-обработчик совершает некоторые преобразования над заявкой и отправляет результат за пределы системы через процесс-шлюз.

Процессы выполняются на двух процессорах, расположенных в разных разъемах на материнской плате физического узла.

Снаружи системы находится симулятор внешнего мира. Он генерирует поток заявок в систему и получает результат обслуживания заявки в системе. Схема взаимодействия процессов в эксперименте представлена на рисунке 13.

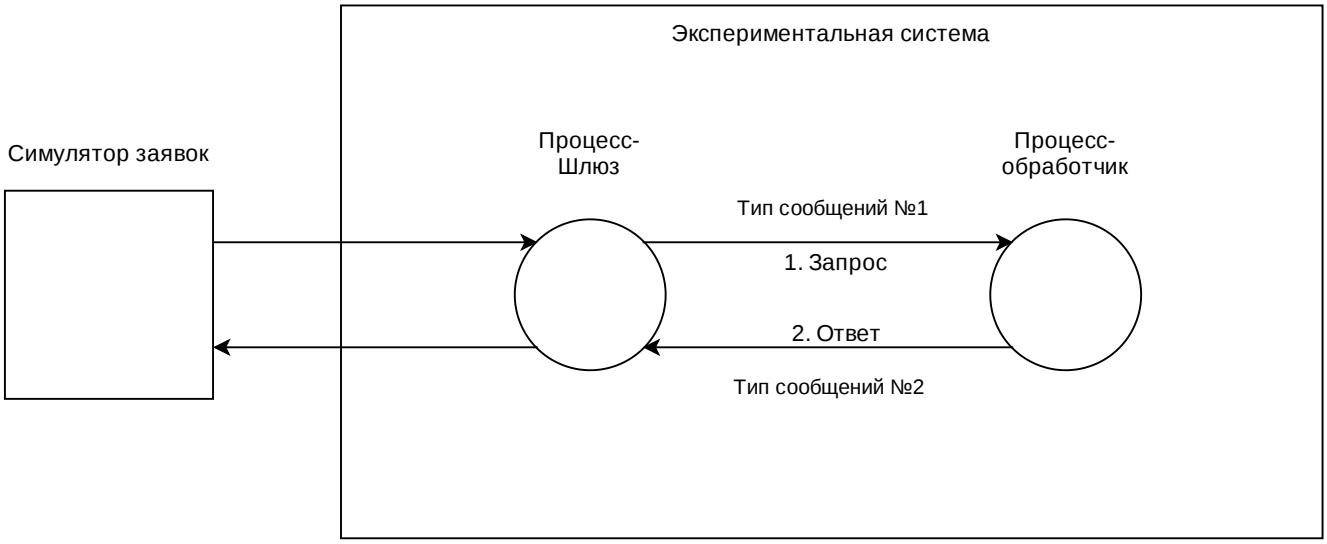

Рисунок 13 – Схема взаимодействия процессов в эксперименте

В настоящей работе замеряется временная задержка на передачу данных между процессами внутри системы, а именно из процесса-шлюза в процесс-обработчик и обратно (сообщения типа №1 и №2 в запросе и ответе между процессом-шлюзом и процессом обработчиком на рисунке 13).

Процессы системы взаимодействуют используют одно соединение, в рамках которого заявки обрабатываются строго последовательно. Обслуживание включает в себя: прием заявки, выполнение пользовательской логики над заявкой и, если необходимо, отправка ответа. Временной задержкой на передачу данных в настоящей работе принимается временной промежуток от начала отправки заявки до **начала обслуживания заявки**. Таким образом, возможна ситуация, когда во время обслуживания очередной заявки процессом в очереди уже находится следующая заявка, временная задержка на передачу которой, таким образом, увеличится на время обслуживания текущей заявки.

Данный сценарий актуален для процесса-обработчика, в котором значительная часть обслуживания заявки осуществляется непосредственно в транспортном потоке, что блокирует получение следующих заявок по соединению. В случае с процессом-шлюзом транспортный поток только читает и диспетчеризует асинхронное обслуживание заявки, т.е. не выполняет обслуживание самой заявки, а потому данная проблема не проявляется.

Пользовательская логика процесса-обработчика в среднем отклоняет 25% заявок, в то время как 75% заявок отправляются в процесс-шлюз.

3.1.3 Используемые обозначения

— Δ – временная задержка между сериями заявок;
— δ – временная задержка между заявками в серии;
— τ – временная задержка на передачу данных;
— T – время обслуживания заявки.
— транспортный поток – поток из пула транспортных потоков, непосредственно обслуживающий полученные заявки.

Из-за сложного характера распределений для представления экспериментальных данных используются гистограммы выборок и 50, 80, 90, 95 и 99 процентили.

3.1.4 Характер экспериментальной нагрузки

Характеристики потока заявок, создаваемого симулятором, приведены в таблице 1.

Таблица 1 – Процентили интервалов между сериями заявок и заявками, создаваемыми симулятором

Процентиль	0%	50%	80%	90%	95%	99%	100%
Δ, мс	5	9	11	12.5	13	15	109
δ, мкс	52	62	82	110	115	128	233

3.1.5 Время обслуживания заявок в процессах

На рисунке 14 представлена гистограмма времени обслуживания заявок в процессе-обработчике. В таблице 2 представлены процентили времени обслуживания заявок в процессе-обработчике.

Таблица 2 – Процентили времени обслуживания заявок в процессе-обработчике

Процентиль	0%	50%	80%	90%	95%	99%	100%
T, мкс	2	11	15	16	17	21	285

На рисунке 15 представлена гистограмма времени обслуживания заявок в процессе-шлюзе. В таблице 3 представлены процентили времени обслуживания заявок в процессе-обработчике.

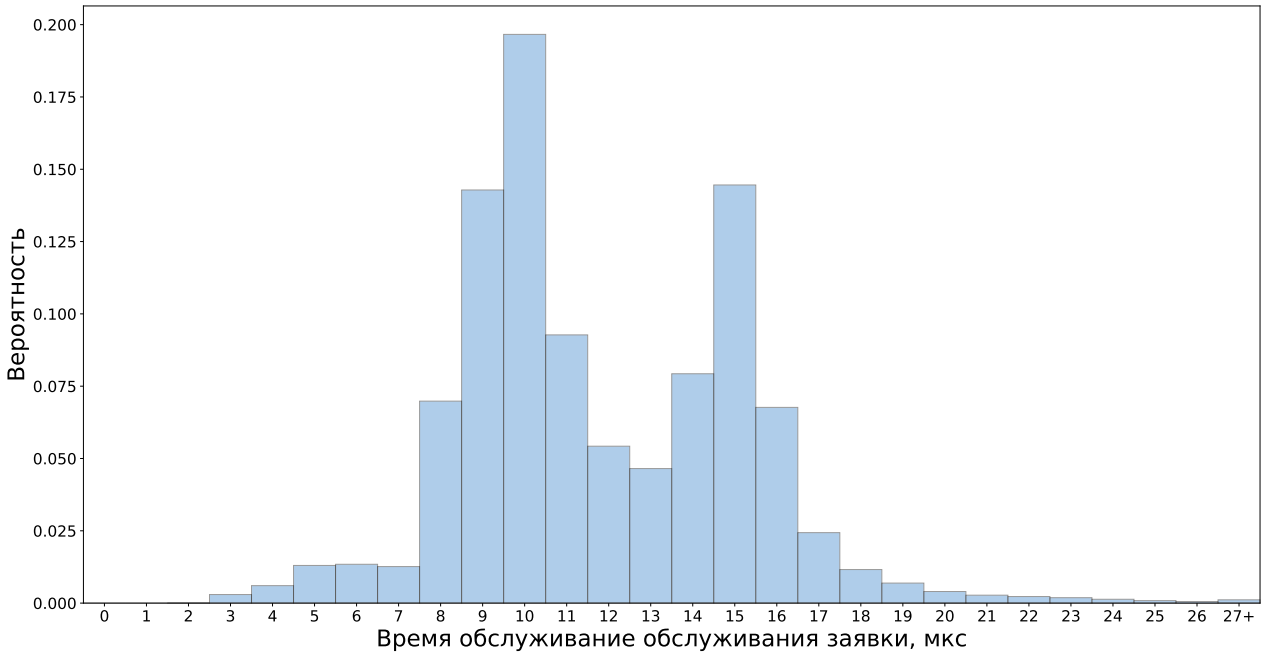

Рисунок 14 – Гистограмма времени обслуживания заявки в процессе-обработчике

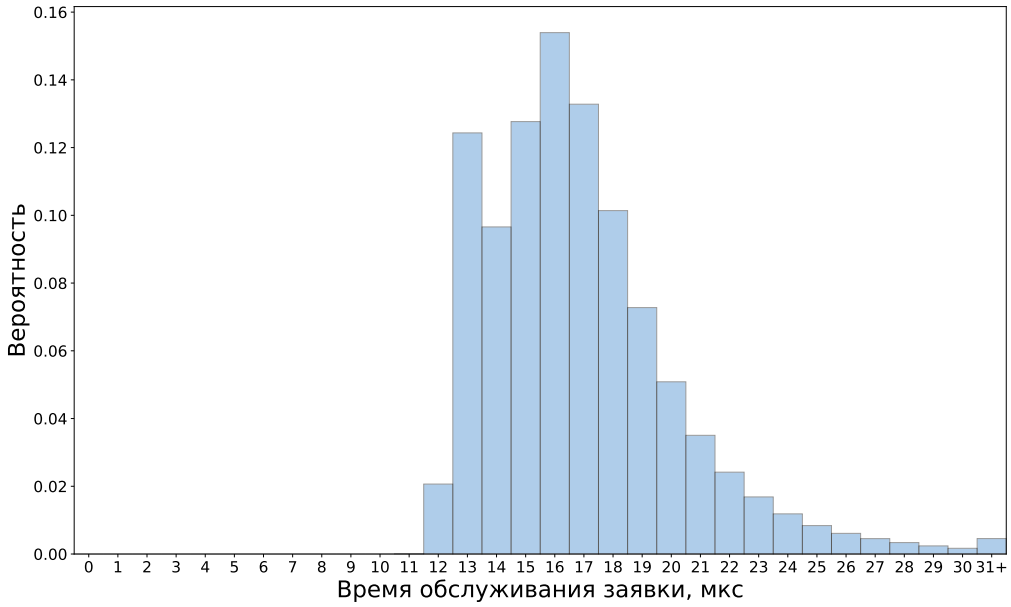

Рисунок 15 – Гистограмма времени обслуживания заявки в процессе-шлюзе

Таблица 3 – Процентили времени обслуживания заявок в процессе-шлюзе

Процентиль	0%	50%	80%	90%	95%	99%	100%
T, мкс	11	16	19	21	23	27	5678

Как было сказано выше, в процессе-шлюзе заявки обслуживания вне транспортного потока, поэтому время обслуживания заявок в процессе-шлюзе не влияет на временную задержку на передачу данных. В случае с процессом-обработчиком значительная часть обслуживания заявки выполняется именно в транспортном потоке, что влияет на временную задержку на передачу данных, т.к. нахождение в очереди на обслуживание влияет на данный показатель.

3.2 Использование TCP для передачи данных

В качестве точки отсчета в настоящей работе выступает метод межпроцессного взаимодействия на основе TCP, используемый посредством сокетов (см. пункт 2.1.1).

Гистограмма временной задержки на передачу данных для данного метода приведена на рисунке 16. В таблице 4 приведены основные временные характеристики данного метода.

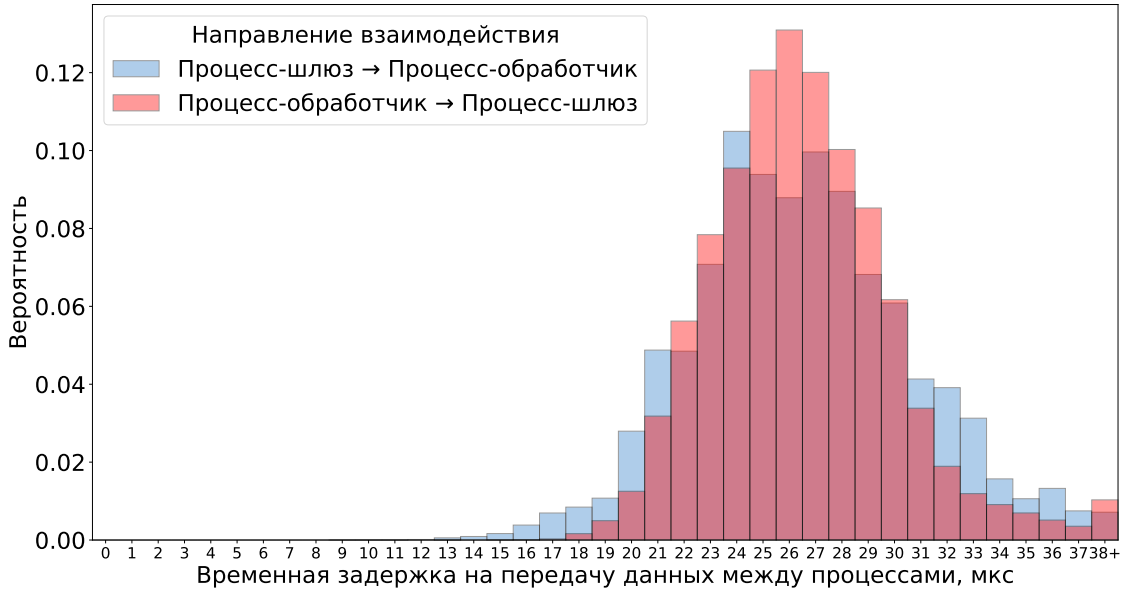

Рисунок 16 – Гистограмма временной задержки на передачу данных между процессами при использовании TCP

Временная задержка на передачу данных в обоих направлениях имеет схожие значения. Это объясняется тем, что накладные расходы на использование TCP через механизм сокетов имеют подавляющее значение над прочими факторами, влияющими на межпроцессное взаимодействие.

Таблица 4 – Основные показатели временной задержки на передачу данных для метода на основе TCP

Направление взаимодействия/ Показатель	Процесс-шлюз → Процесс-обработчик	Процесс-обработчик → Процесс-шлюз
min(τ), мкс	9	13
50%, мкс	26	26
80%, мкс	30	29
90%, мкс	32	30
95%, мкс	34	32
99%, мкс	37	38
max(τ), мс	2.1	9.2

3.3 Использование разделяемой памяти для передачи данных

3.3.1 Использование TCP для оповещения о появлении данных

В данном подразделе приведены данные об экспериментах с методом межпроцессного взаимодействия, описанным в пункте 2.3.2.

В таблице 5 приведены основные временные характеристики данного метода. На рисунке 17 приведена гистограмма временной задержки на передачу данных для данного метода.

Таблица 5 – Основные показатели временной задержки на передачу данных для метода, использующего разделяемую памяти для передачи данных и TCP для оповещения о появлении данных в ней

Направление взаимодействия/ Показатель	Процесс-шлюз → Процесс-обработчик	Процесс-обработчик → Процесс-шлюз
$min(\tau)$, мкс	1	2
50%, мкс	27	27
80%, мкс	30	29
90%, мкс	32	30
95%, мкс	34	31
99%, мкс	36	35
$max(\tau)$, мс	3	9.8

Как описано выше, значительная часть обслуживания заявки процессом-обработчиком происходит непосредственно в транспортном потоке. Из-за этого к моменту конца обработки текущей заявки очередная заявка уже может находить-

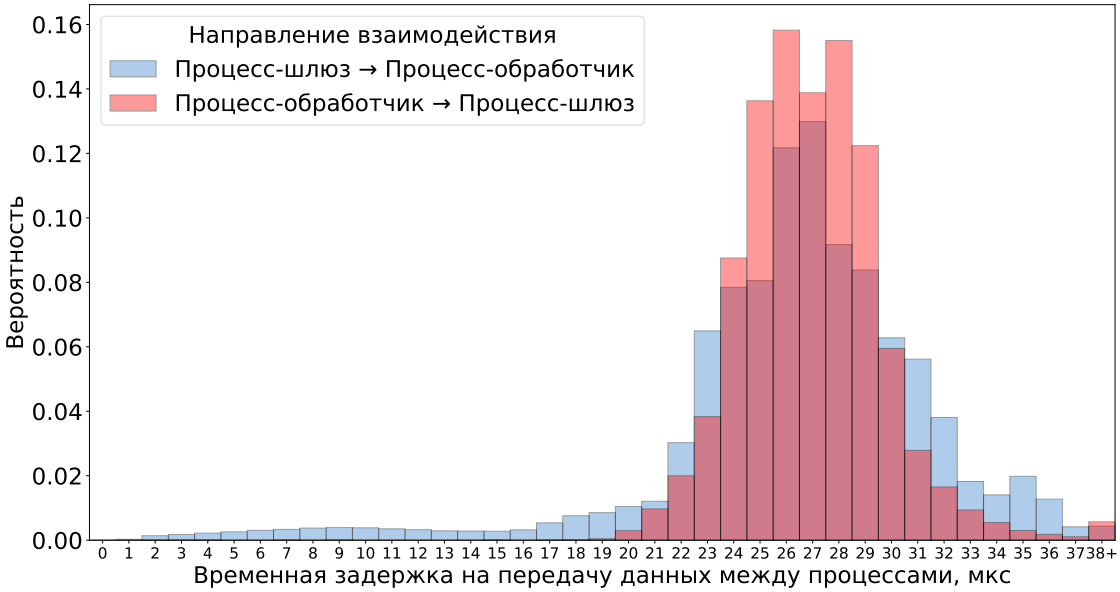

Рисунок 17 – Гистограмма временной задержки на передачу данных между процессами при использовании разделяемой памяти для передачи данных и TCP для оповещения о появлении данных в ней

ся в очереди в разделяемой памяти, что позволяет использовать оптимизацию, описанную в подразделе 2.2. А именно принять и начать обслуживание очередной заявки, не используя дорогостоящий механизм оповещения, если заявка уже находится в очереди в разделяемой памяти. Этот эффект хорошо заметен для минимальной временной задержки на межпроцессное взаимодействие, а также на гистограмме.

Прием заявки процессом-шлюзом не связан с обслуживанием заявки, поэтому к моменту, когда процесс-шлюз заканчивает прием и диспетчеризацию заявки, очередь входящих заявок в данном эксперименте пуста и поток-шлюз переходит к пассивному ожиданию новых заявок. Таким образом, приему и обслуживанию большинства заявок сопутствует пассивное ожидание сигнала по TCP, что негативно сказывается на временной задержке на передачу данных.

3.3.2 Использование мультиплексора в разделяемой памяти для оповещения о появлении данных

В данном подразделе приведены данные об экспериментах с семейством методов межпроцессного взаимодействия, описанными в пункте 2.3.3.

3.3.2.1 Методы с пассивным ожиданием оповещений

В методах с пассивным ожиданием оповещений поток мультиплексора событий использует примитив *futex* для ожидания новых сигналов (см. подпункты 2.3.4.2 и 2.3.4.3). Поток процесса-читателя, опрашивающий мультиплексор в разделяемой памяти, находится в состоянии сна и пробуждается процессом-писателем при необходимости.

Диспетчеризация и обработка соединений по модели ”Полусинхронный/Полуреактивный” В данном параграфе приведены данные об экспериментах с методом межпроцессного взаимодействия, описанным в подпункте 2.3.4.2.

В таблице 6 приведены основные временные характеристики данного метода. На рисунке 18 приведена гистограмма временной задержки на передачу данных для данного метода.

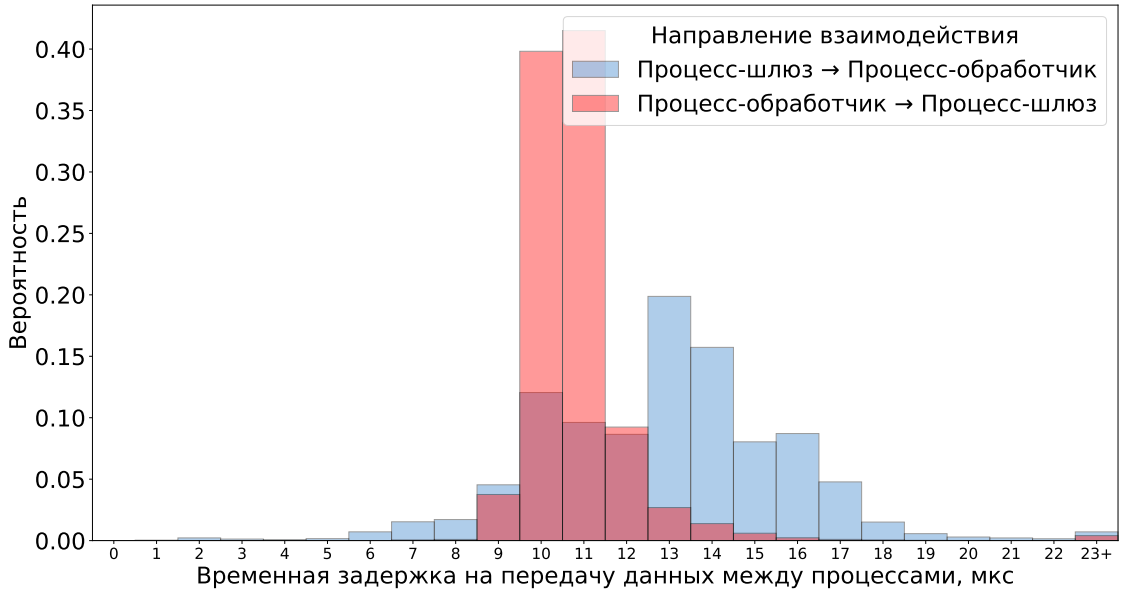

Рисунок 18 – Гистограмма временной задержки на передачу данных между процессами для метода, использующего разделяемую память для передачи данных, пассивное ожидание событий на мультиплексоре событий в разделяемой памяти и метод ”Полусинхронный/Полуреактивный“ при обслуживании заявок

Использовании более эффективного метода межпроцессного взаимодействия наглядно показывает эффект от влияния времени обслуживания заявки

Таблица 6 – Основные показатели временной задержки на передачу данных между процессами для метода, использующего разделяемую память для передачи данных, пассивное ожидание оповещений в мультиплексоре событий в разделяемой памяти и метод ”Полусинхронный/Полуреактивный“ при обслуживании заявок

Направление взаимодействия/ Показатель	Процесс-шлюз → Процесс-обработчик	Процесс-обработчик → Процесс-шлюз
min(t), мкс	1	3
50%, мкс	13	11
80%, мкс	15	11
90%, мкс	16	12
95%, мкс	17	13
99%, мкс	21	15
max(t), мс	6.9	11.6

в транспортном потоке на временную задержку на передачу данных. Процесс-шлюз с минимальной временной задержкой принимает и диспетчеризует заявку для дальнейшей обработки и сразу же готов принимать следующую заявку. Процесс-обработчик же использует транспортный поток для частичного обслуживания заявки (см. рисунок 14), а потому в среднем временная задержка на передачу данных имеет худшие показатели, так как это может задерживать прием очередных заявок.

Диспетчеризация и обработка соединений по модели ”Лидер/Последователи”

В данном параграфе приведены данные об экспериментах с методом межпроцессного взаимодействия, описанным в подпункте 2.3.4.3.

В таблице 7 приведены основные временные характеристики данного метода. На рисунке 19 приведена гистограмма временной задержки на передачу данных для данного метода.

По сравнению с методом ”Полусинхронный/Полуреактивный“ данный метод ожидаемо имеет меньшую временную задержку на передачу данных (см. подпункт 2.3.4.3). В данном методе поток, получивший оповещение из мультиплексора оповещений, приступит к обслуживанию соединений сразу после того, как пробудит следующий поток-лидер. В то время как для предыдущего метода обработка заявки начнется только после пробуждения отдельного потока и постановки его на выполнение ОС.

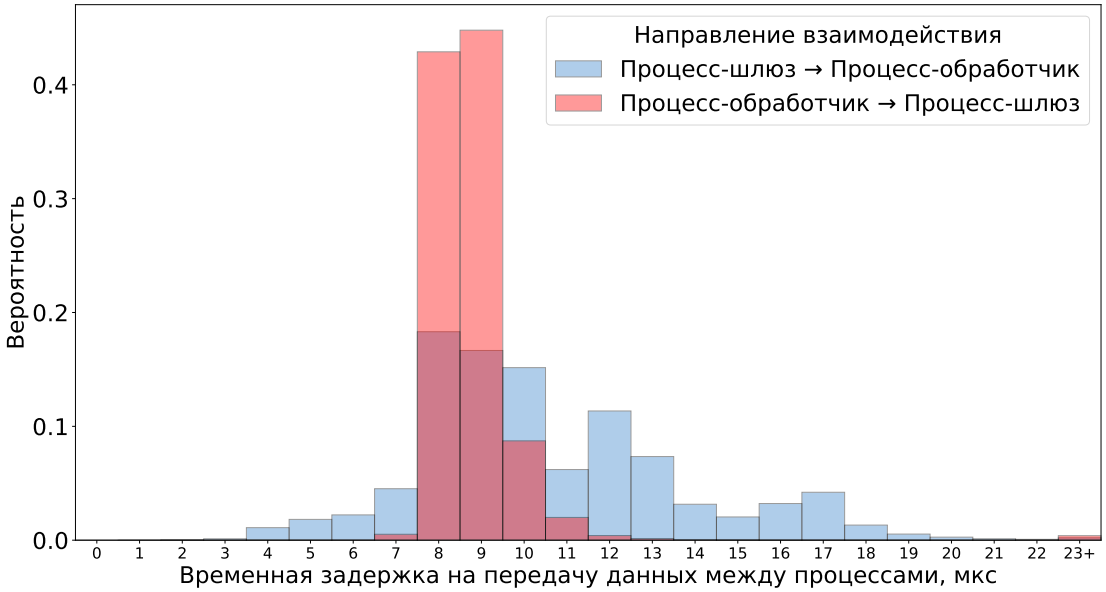

Рисунок 19 – Гистограмма временной задержки на передачу данных между процессами для метода, использующего разделяемую память для передачи данных, пассивное ожидание оповещений в мультиплексоре событий в разделяемой памяти и метод ”Лидер/Последователи“ при обслуживании заявок

Таблица 7 – Основные показатели временной задержки на передачу данных между процессами для метода, использующего разделяемую память для передачи данных, пассивное ожидание оповещений в мультиплексоре событий в разделяемой памяти и метод ”Лидер/Последователи“ при обслуживании заявок

Направление взаимодействия/ Показатель	Процесс-шлюз → Процесс-обработчик	Процесс-обработчик → Процесс-шлюз
min(t), мкс	1	2
50%, мкс	10	9
80%, мкс	13	9
90%, мкс	15	10
95%, мкс	17	10
99%, мкс	19	12
max(t), мс	2.4	9.5

Выводы по исследованиям методов с пассивным ожиданием оповещений

Методы оповещения процессов о появлении данных в разделяемой памяти с использованием мультиплексора в разделяемой памяти показывают существенно меньшую временную задержку на доставку оповещения, чем метод с использованием TCP. Это объясняется существенно меньшим количеством системных вы-

зовов для обеих сторон взаимодействия и меньшей временной задержкой на исполнение оставшихся системных вызовов.

Полученный результат подтверждает тезисы автора этих методов о превосходстве метода ”Лидер/Последователи“ над методом ”Полусинхронный/Полуреактивный“ при отсутствии необходимости приоритезации обработки заявок [25, с. 398]. Однако, в отличие от исходного метода ”Полусинхронный/Полуреактивный“ [25, с. 375], работающего с сокетами и системным мультиплексором событий, в текущей ситуации нет необходимости считывать данные из сокета, чтобы переложить их в централизованную очередь для последующей обработки. Поэтому преимущество не такое существенное.

3.3.2.2 Метод с активным ожиданием оповещений

В методе с активным ожиданием оповещений поток мультиплексора событий находится в режиме постоянного опроса первого уровня мультиплексора на предмет оповещений (см. подпункт 2.3.4.4). В данном подпункте рассматривается исключительно метод обслуживания заявок ”Лидер/Последователи“ т.к. в параграфе выше он показал лучший результат по сравнению с методом обслуживания заявок ”Полусинхронный/Полуреактивный“, а также исходя из анализа применимости метода активного ожидания оповещений этим методам обслуживания соединений (см. подпункт 2.3.4.4).

Диспетчеризация и обработка соединений по модели ”Лидер/Последователи”
В данном параграфе приведены данные об экспериментах с методом межпроцессного взаимодействия, описанным в подпункте 2.3.4.4.

В таблице 8 приведены основные временные характеристики данного метода. На рисунке 20 приведена гистограмма временной задержки на передачу данных для данного метода.

Выводы по исследованию метода с активным ожиданием Метод с активным ожиданием событий в мультиплексоре в разделяемой памяти ожидаемо показывает лучший результат по сравнению со всеми ранее рассмотренными методами. Это происходит потому что в данном случае нет необходимости пробуждать и дожидаться пробуждения потока для получения оповещения о появлении данных в очереди. Исходя из разницы временной задержки на передачу данных в методах

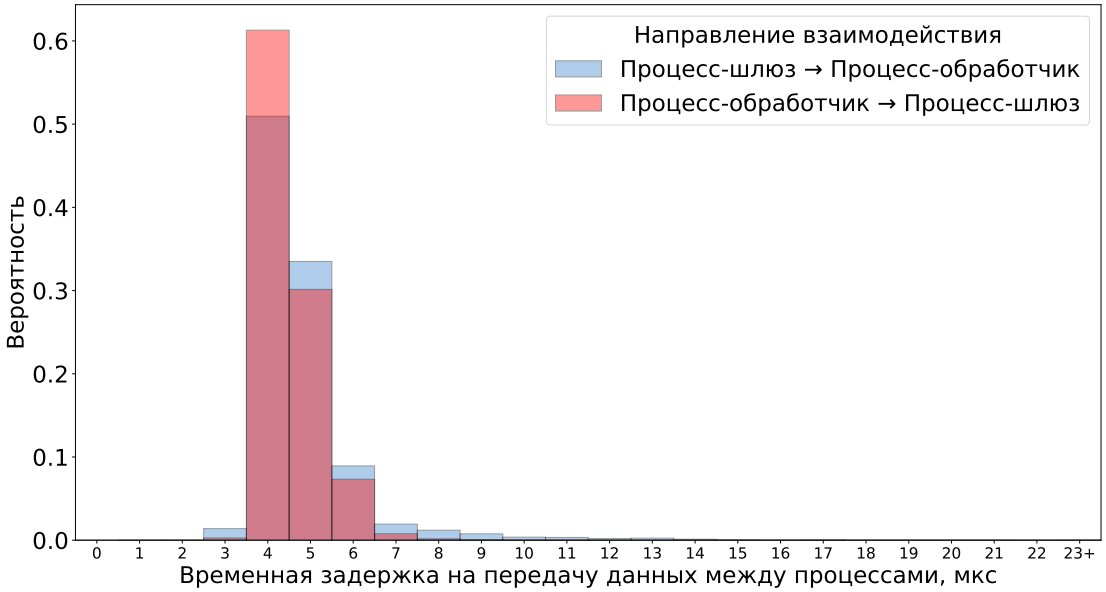

Рисунок 20 – Гистограмма временной задержки на передачу данных между процессами для метода, использующего разделяемую память для передачи данных, активно опрашиваемый мультиплексор в разделяемой памяти и модель ”Лидер/Последователи“ при обслуживании заявок

Таблица 8 – Основные показатели временной задержки на передачу данных между процессами для метода, использующего разделяемую память для передачи данных, активно опрашиваемый мультиплексор в разделяемой памяти и модель ”Лидер/Последователи“ при обслуживании заявок

Направление взаимодействия/ Показатель	Процесс-шлюз → Процесс-обработчик	Процесс-обработчик → Процесс-шлюз
min(t), мкс	1	3
50%, мкс	4	4
80%, мкс	5	5
90%, мкс	6	5
95%, мкс	7	6
99%, мкс	10	6
max(t), мс	0.064	0.017

с активным и пассивным ожиданием, пробуждение потока до постановки его на выполнение занимает около 5-6 мкс. В работе другого автора [17] была измерена временная задержка на пробуждение потоков, прикрепленных к разным ядрам разных процессоров, посредством системного вызова futex. Измерения проводились на процессоре AMD Opteron 6272 и показали $\mu = 24640.5$ машинных циклов

от системного вызова до выполнения первой инструкции пробужденным потоком. Или ≈ 11 мкс при частоте процессора 2.1 ГГц, что похоже на обозначенный выше результат с учетом использования в настоящей работе более современного аппаратного обеспечения.

Для данного метода распределение временной задержки на передачу данных от процесса-обработчика к процессу-шлюзу схоже с предыдущими методами, но этот показатель существенно отличается при передаче от процесса-шлюза к процессу-обработчику. Это может быть вызвано тем, что более быстрый метод межпроцессного взаимодействия при данной постановке эксперимента приводит к отсутствию очередей и, как следствие, время обслуживания заявки в транспортном потоке процесса-обработчика не влияет на временную задержку на передачу данных.

Данный подход обладает существенным **недостатком**. Поскольку поток, активно опрашивающий мультиплексор, исполняется до получения и обработки события, то существует возможность, что планировщик ОС вытеснит этот поток с процессора по окончании выделенного ему кванта процессорного времени. Размер кванта в Linux расчитывается динамически. Если событие не будет получено и обработано за это время, то поток может быть вытеснен с процессора и временная задержка на передачу данных увеличится на неопределенное время. Данный недостаток не проявляется в проведенном эксперименте, поскольку интервал Δ между сериями заявок мал ($\approx$ 10мс), а сервер имеет достаточно свободных ресурсов.

Выводы по главе 3

Проведено экспериментальное сравнение разработанных методов межпроцессного взаимодействия.

а) Методы межпроцессного взаимодействия, использующие мультиплексор в разделяемой памяти для оповещения о появлении данных в очереди в разделяемой памяти имеют существенно меньшую временную задержку на передачу данных, чем метод, использующий для этого TCP. А именно, *17 мкс* и *10 мкс* для 95 процентиля для пассивного варианта ”Лидер/Последователи“ против *34 мкс* и *31 мкс* для 95 процентиля для TCP с передачей данных через очередь в разделяемой памяти;

б) в семействе пассивных методов межпроцессного взаимодействия на основе мультиплексора в разделяемой памяти наименьшую временную задержку на передачу данных показала вариация с использованием метода обслуживания соединений ”Лидер/Последователи“, а именно *17 мкс* и *10 мкс* для 95 процентиля против *17 мкс* и *13 мкс* для 95 процентиля при использовании метода обслуживания соединений ”Полусинхронный/Полуреактивный“;

в) самой низкой временной задержки на передачу данных удалось добиться при использовании активно опрашивающей мультиплексор вариации метода межпроцессного взаимодействия, использующего метод ”Лидер/Последователи“ при обслуживании соединений. А именно, *7 мкс* и *6 мкс* для 95 процентиля;

г) при использовании пассивных методов на основе мультиплексора в разделяемой памяти заметны эффекты от обслуживания соединений в транспортном потоке на временную задержку на передачу данных. В проведенном эксперименте в процессе-шлюзе заявки частично обслуживаются именно в транспортном потоке, из-за чего могут образовываться очереди и увеличиваться временная задержка на передачу данных. В методе активного опроса мультиплексора этот эффект не заметен, так как временная задержка на передачу данных меньше, соответственно, меньше временная задержка реакции на очередную заявку и, следовательно, меньше возможностей для формирования очереди из заявок в разделяемой памяти;

д) метод активного опроса мультиплексора обладает недостатком. Поток, активно опрашивающий мультиплексор в разделяемой памяти, может быть вытеснен с процессора по окончании отведенного ему кванта процессорного времени. Размер кванта в Linux расчитывается динамически. В проведенном эксперименте данный эффект не наблюдается, так как серии заявок отправляются симулятором в систему часто, в среднем каждые 10 миллисекунд. Сервер при этом имеет достаточно свободных ресурсов. Данный недостаток может быть необходимо разрешить для обеспечения должного уровня качества обслуживания соединений в системе.

ЗАКЛЮЧЕНИЕ

Целью настоящей работы являлось уменьшение временной задержки на передачу данных между процессами в пределах одного физического узла путем разработки и применения методов эффективного межпроцессного взаимодействия. Было выполнено сравнение методов межпроцессного взаимодействия и синхронизации. Исследованы подходы других авторов. Разработаны и реализованы новые методы межпроцессного взаимодействия. Проведено экспериментальное исследование разработанных методов межпроцессного взаимодействия.

Результатом работы стало семейство новых методов межпроцессного взаимодействия. Передача данных в них осуществляется через очередь в разделяемой памяти. Оповещение о появлении данных в очереди осуществляется через разработанный и реализованный мультиплексор оповещений в разделяемой памяти. Он позволяет производить большую часть операций по оповещению процесса-читателя в пользовательской памяти, а ядро использовать только при необходимости процессу-читателю ожидать оповещений в режиме сна и процессу-писателю разбудить его.

Были реализованы различные методы обслуживания соединений по полученным из мультиплексора оповещениям, исследовано влияние активного опроса мультиплексора на временную задержку на передачу данных. Исходя из полученных результатов, разработанные в данной работе методы показали существенно меньшую временную задержку на передачу данных по сравнению с методами, использующими TCP. Метод обслуживания соединений ”Лидер/Последователи“ при использовании с мультиплексором оповещений в разделяемой памяти показывает меньшую временную задержку на передачу данных, чем метод обслуживания соединений ”Полусинхронный/Полуреактивный“. Применение метода активного опроса мультиплексора позволило существенно уменьшить временную задержку на передачу данных по сравнению с аналогичным методом с пассивным ожиданием оповещений в режиме сна.

Основные результаты работы представлены на IX Конгрессе Молодых Ученых. Результаты работы использованы в программной платформе для высокочастотной алгоритмической торговли Tbricks компании Itiviti.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 *Губарев В. Ю.* Реализация методов эффективного взаимодействия процессов в распределенных системах [Электронный ресурс] // Сборник тезисов докладов конгресса молодых ученых. Электронное издание. — Университет ИТМО, 2020. — URL: `https://kmu.itmo.ru/file/download/application/9365` (дата обращения: 23.05.2020).

2 *Губарев В. Ю.*, *Косяков М. С.* Разработка и реализация методов эффективного взаимодействия процессов в распределенных системах [Электронный ресурс] // Сборник тезисов докладов конгресса молодых ученых. Электронное издание. — 2018. — URL: `http://openbooks.ifmo.ru/ru/file/7292/7292.pdf` (дата обращения: 22.04.2020).

3 *Кузичкина А. О.* Исследование и разработка методов трассировки событий в параллельных и распределенных системах : Выпускная квалификационная работа / Кузичкина Анастасия Олеговна. — Факультет ПИиКТ : Университет ИТМО, 2017.

4 *Blieberger J.*, *Burgstaller B.*, *Scholz B.* Busy Wait Analysis // Reliable Software Technologies — Ada-Europe 2003 / ed. by J.-P. Rosen, A. Strohmeier. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2003. — P. 142–152. — ISBN 978-3-540-44947-8.

5 *Buntinas D.*, *Mercier G.*, *Gropp W.* Implementation and Shared-Memory Evaluation of MPICH2 over the Nemesis Communication Subsystem // Recent Advances in Parallel Virtual Machine and Message Passing Interface / ed. by B. Mohr [et al.]. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2006. — P. 86–95. — ISBN 978-3-540-39112-8.

6 *Chai L.*, *Hartono A.*, *Panda D. K.* Designing High Performance and Scalable MPI Intra-node Communication Support for Clusters // 2006 IEEE International Conference on Cluster Computing. — 2006. — P. 1–10.

7 Comparing and Evaluating epoll, select, and poll Event Mechanisms / L. Gammo [et al.] // Proceedings of the 6th Annual Ottawa Linux Symposium. — 2004. — Vol. 1. — P. 215–225.

8 *Corporation I.* Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2 [Электронный ресурс]. — 2016. — URL: https://www.intel.ru/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.pdf (дата обращения: 10.02.2020).

9 Draft: Have you checked your IPC performance lately? [Электронный ресурс] / S. Smith [et al.]. — 2012. — URL: https://www.semanticscholar.org/paper/Draft-%3A-Have-you-checked-your-IPC-performance-Smith-Madhavapeddy/1cb3b82e2ef6a95b576573b8af0f3ec6f7bc21d9 (дата обращения: 25.03.2020).

10 *Drepper U.* Futexes Are Tricky [Электронный ресурс]. — 2004. — June. — URL: https://dept-info.labri.fr/~denis/Enseignement/2008-IR/Articles/01-futex.pdf (дата обращения: 23.04.2020).

11 Efficient Shared Memory Message Passing for Inter-VM Communications / F. Diakhaté [et al.] // Euro-Par 2008 Workshops - Parallel Processing / ed. by E. César [et al.]. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2009. — P. 53–62. — ISBN 978-3-642-00955-6.

12 Framework for Scalable Intra-Node Collective Operations using Shared Memory / S. Jain [et al.] // SC18: International Conference for High Performance Computing, Networking, Storage and Analysis. — 2018. — P. 374–385.

13 *Franke H., Russell R., Kirkwood M.* Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux [Электронный ресурс]. — 2002. — Jan. — URL: https://www.kernel.org/doc/ols/2002/ols2002-pages-479-495.pdf (дата обращения: 23.04.2020).

14 *Ganjaliyev F.* Spin-Then-Sleep: A Machine Learning Alternative to Queue-based Spin-then-Block Strategy // International Journal of Advanced Computer Science and Applications. — 2019. — Vol. 10, no. 3. — DOI: https://doi.org/10.14569/IJACSA.2019.0100377.

15 *Gaztanaga I.* Chapter 18. Boost.Interprocess [Электронный ресурс]. — URL: https://www.boost.org/doc/libs/1_63_0/doc/html/interprocess.html (дата обращения: 25.03.2020).

16 *Gulati M.* Reducing the Inter-Process Communication Time on Local Host by Implementing Seamless Socket Like, ”low latency“ Interface Over Shared Memory : Master’s thesis / Gulati Mauli. — Department of Computer Science & Engineering : Thapar University, 2009.

17 *Hale K. C.*, *Dinda P. A.* An Evaluation of Asynchronous Software Events on Modern Hardware // 2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS). — 2018. — P. 355–368.

18 *Hammar A.* Analysis and Design of High Performance Inter-core Process Communication for Linux : Master’s thesis / Hammar Andreas. — Uppsala University, Department of Information Technology, 2014. — UPTEC IT, ISSN 1401-5749 ; 14 020.

19 *Huang A. S.*, *Olson E.*, *Moore D. C.* LCM: Lightweight Communications and Marshalling // 2010 IEEE/RSJ International Conference on Intelligent Robots and Systems. — 2010. — P. 4057–4062.

20 *Informatica*. Ultra messaging [Электронный ресурс]. — URL: `https://www.informatica.com/products/data-integration/ultra-messaging.html` (дата обращения: 20.05.2020).

21 *Khartchenko E.* Optimizing Computer Applications for Latency: Part 1: Configuring the Hardware [Электронный ресурс]. — 2017. — URL: `https://software.intel.com/content/www/us/en/develop/articles/optimizing-computer-applications-for-latency-part-1-configuring-the-hardware.html` (дата обращения: 28.03.2020).

22 Leader/followers / D. C. Schmidt [et al.] // Proceeding of the 7th Pattern Languages of Programs Conference. — 2000. — P. 1–40.

23 LTTng: an open source tracing framework for Linux [Электронный ресурс]. — URL: `https://lttng.org/` (дата обращения: 25.03.2020).

24 *Macdonell A. C.* Shared-Memory Optimizations for Virtual Machines : PhD dissertation / Macdonell A. Cameron. — Department of Computing Science : University of Alberta, 2011. — DOI: `https://doi.org/10.7939/R3Q715`.

25 Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects. Vol. 2 / D. C. Schmidt [et al.]. — John Wiley & Sons, 2013.

26 Real-time inter-process communication in heterogeneous programming environments / A. Alexandrescu [et al.] // 2016 20th International Conference on System Theory, Control and Computing (ICSTCC). — 2016. — P. 283–288.

27 *Schmidt D. C.* Acceptor and connector // Pattern Languages of Program Design. — 1996. — Vol. 3. — P. 191–229.

28 *Schmidt D. C.*, *Cranor C. D.* Half-Sync/Half-Async // Second Pattern Languages of Programs, Monticello, Illinois. — 1995.

29 *Schmidt D. C.* Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects. — Wiley, 2000. — P. 633. — ISBN 0471606952.

30 Shared-Memory Optimizations for Inter-Virtual-Machine Communication / Y. Ren [et al.] // ACM Comput. Surv. — New York, NY, USA, 2016. — Feb. — Vol. 48, no. 4. — ISSN 0360-0300. — DOI: https://doi.org/10.1145/2847562.

31 Shimmy: Shared Memory Channels for High Performance Inter-Container Communication [Электронный ресурс] / M. Abranches [et al.] // 2nd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 19). — Renton, WA : USENIX Association, 07.2019. — URL: https://www.usenix.org/conference/hotedge19/presentation/abranches (дата обращения: 02.03.2020).

32 *Software T.* TIBCO® Messaging [Электронный ресурс]. — URL: https://www.tibco.com/products/tibco-messaging (дата обращения: 20.05.2020).

33 *Solace*. Messaging Middleware for Event-Driven Enterprises [Электронный ресурс]. — URL: https://solace.com/messaging-middleware/ (дата обращения: 20.05.2020).

34 *Strebelow R.*, *Tribastone M.*, *Prehofer C.* Performance Modeling of Design Patterns for Distributed Computation // 2012 IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems. — 2012. — P. 251–258.

35 *Strebelow R.*, *Prehofer C.* Analysis of Event Processing Design Patterns and Their Performance Dependency on I/O Notification Mechanisms // Multicore Software Engineering, Performance, and Tools / ed. by V. Pankratius, M. Philippsen. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2012. — P. 54–65. — ISBN 978-3-642-31202-1.

36 TZC: Efficient Inter-Process Communication for Robotics Middleware with Partial Serialization / Y.-p. Wang [et al.] // 2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). — 2019. — P. 7805–7812.

37 *Venkataraman A.*, *Jagadeesha K. K.* Evaluation of inter-process communication mechanisms [Электронный ресурс]. — URL: http://pages.cs.wisc.edu/adityav/Evaluation_of_Inter_Process_Communication_Mechanisms.pdf (дата обращения: 02.05.2020).

38 *Wei Huang*, *Koop M. J.*, *Panda D. K.* Efficient one-copy MPI shared memory communication in Virtual Machines // 2008 IEEE International Conference on Cluster Computing. — 2008. — P. 107–115.

39 *Xiaodi K.* Interprocess Communication Mechanisms With Inter-Virtual Machine Shared Memory : Master's thesis / Xiaodi Ke. — Department of Computing Science : University of Alberta, 2011. — DOI: https://doi.org/10.7939/R3PH6H.

40 *Zhang Q.*, *Liu L.* Workload Adaptive Shared Memory Management for High Performance Network I/O in Virtualized Cloud // IEEE Transactions on Computers. — 2016. — Vol. 65, no. 11. — P. 3480–3494.

ПРИЛОЖЕНИЕ А. ИСХОДНЫЙ КОД АЛГОРИТМОВ ПОЛУЧЕНИЯ ОПОВЕЩЕНИЙ В МУЛЬТИПЛЕКСОРЕ В РАЗДЕЛЯЕМОЙ ПАМЯТИ

Листинг А.1 – Исходный код процедуры получения оповещений из мультиплексора событий в разделяемой памяти

```
void MutliplexerServer::handle_signals() {
    // Шаг 1. Если в futex записан 0, значит, нет оповещений для
   обработки. Тогда процесс переходит в состояние сна.
    m_mux->wait();
    // Шаг 2. Атомарно получить актуальное значение futex и
   установить вместо него 0.
    int32_t futex = atomic_exchange(&m_futex, 0);
    // Шаг 3. Подсчитать количество установленных битов в числе,
   чтобы не выполнять линейное сканирование всех 32 битов.
    uint8_t cnt = popcnt(futex);
    for (uint8_t i = 0; i < cnt; i++) {
        // Шаг 4. Для каждого бита futex проверить соответствующие
   ему сигнальные числа.
        uint8_t f = get_unset_lsb(&futex);
        // Шаг 5. Атомарно получить значение сигнального числа и
   записать в него 0.
        int64_t signal = atomic_exchange(&m_signal[f], 0);
        uint8_t nsignals = popcntl(signal);
        // Шаг 6. Для каждого найденного сигнала запустить его
   обработку.
        for (uint8_t j = 0; j < nsignals; j++) {
            uint8_t s = get_unset_lsb(&signal);
            this->handle_signal(i * 64 + s);
        }
    }
}

// Ожидание новых оповещений на futex в режиме сна или методом
   активного опроса мультиплексора.
void Multiplexer::wait();
// Выполняет обработку соединения, которому ранее был выдан номер id
void MultiplexerServer::handle_signal(Signal id);

// Возвращает количество выставленных битов в числе
uint8_t popcnt(int32_t value);
uint8_t popcntl(int64_t value);

// Сбрасывает младший бит числа и возвращает позицию этого бита.
uint8_t get_unset_lsb(uint32_t & value);
uint8_t get_unset_lsb(uint64_t & value);
```

Листинг А.2 – Исходный код процедуры получения оповещений из мультиплексора событий в разделяемой памяти для метода обслуживания соединений ”Лидер/Последователи“

```
void MutliplexerServer::handle_signals() {
    // Шаг 1. Если в futex записан 0, значит, нет оповещений для
   обработки. Тогда процесс переходит в состояние сна.
    m_mux->wait();
    int32_t futex = atomic_exchange(&m_futex, 0);
    std::vector<int32_t> signals;
    listeners.reserve(Multiplexer::c_signals_per_mux);
    uint8_t cnt = popcnt(futex);
    for (uint8_t i = 0; i < cnt; i++) {
        uint8_t f = get_unset_lsb(&futex);
        int64_t signal = atomic_exchange(&m_signal[f], 0);
        uint8_t nsignals = popcntl(signal);
        // Шаг 6. Для каждого найденного сигнала запустить его
   обработку.
        for (uint8_t j = 0; j < nsignals; j++) {
            uint8_t s = get_unset_lsb(&signal);
            // Для каждого полученного оповещения
            // отметить соединение как Handling,
            // либо проигнорировать оповещение
            if (this->should_handle(i * 64 + s)) {
                signals.emplace_back(i * 64 + s);
            }
        }
    }

    // Создать нового лидера, который будет
    // выполнять процедуру handle_signals следующим
    m_thread_pool->promote_new_leader();

    // Непосредственно обслуживание соединений по полученным
   оповещениям
    for (int32_t signal : signals) {
        this->handle_signal(i * 64 + s);
    }
}

// Возвращает true, если соединение было Idle, и помечает его
   Handling
// Если соединение в состоянии Handling, то переводит его в
   состояние KeepHandling
// Возвращает false, если любое другое состояние.
bool MultiplexerServer::should_handle(Signal id);
```