УДК 004.43

В. Е. Жижин

Инженер-программист
ИЦ ППВО

СИНХРОНИЗАЦИЯ И КОНКУРЕНТНОСТЬ В RUST

В статье рассматриваются основные средства синхронизации в многопоточном программировании и особенности их реализации в языке *Rust*. Цель исследования состоит в выявлении специфики подхода *Rust* к организации конкурентного доступа к данным. Методологическую основу работы составили анализ литературы по многопоточному программированию и сравнительное рассмотрение классических примитивов синхронизации. Установлено, что *Rust* использует традиционные механизмы синхронизации, однако встраивает их в систему владения, заимствования и типовых ограничений *Send* и *Sync*. Показано, что такой подход позволяет обнаруживать значительную часть ошибок конкурентного доступа на этапе компиляции, делает модель разделяемого состояния более прозрачной и повышает надежность программных систем. Сделан вывод о перспективности *Rust* для разработки безопасного многопоточного программного обеспечения.

Ключевые слова: многопоточность, синхронизация, конкурентность, *Rust*, мьютекс, атомарные операции, каналы сообщений.

V. E. Zhizhin

Software Engineer
ITC PPPVO

SYNCHRONIZATION AND CONCURRENCY IN RUST

The article examines the main synchronization tools used in multithreaded programming and the specifics of their implementation in the Rust programming language. The aim of the study is to identify the distinctive features of Rust's approach to organizing concurrent data access. The methodological basis of the work includes the analysis of literature on multithreaded programming and a comparative examination of classical synchronization primitives. It is established that Rust employs traditional synchronization mechanisms, but integrates them into the ownership and borrowing system as well as into the Send and Sync type constraints. The paper shows that this approach makes it possible to detect a significant part of concurrent access errors at compile time, makes the shared-state model more transparent, and increases the reliability of software systems. It is concluded that Rust is a promising tool for developing safe multithreaded software.

Keywords: multithreading, synchronization, concurrency, Rust, mutex, atomic operations, message passing.

Введение

Многопоточное программирование является одним из основных средств повышения производительности современных программных систем. Распространение многоядерных архитектур сделало параллельное выполнение вычислений стандартным требованием для системного, сетевого и прикладного программного обеспечения. Вместе с тем рост степени параллелизма сопровождается увеличением числа ошибок, связанных с совместным доступом к данным: состояний гонки, взаимных блокировок, нарушения согласованности структур и иных трудно воспроизводимых дефектов [1].

В традиционных языках программирования средства синхронизации обычно реализуются в виде библиотечных или плат-

форменных механизмов, а корректность их применения в значительной степени зависит от дисциплины разработчика. В результате даже работоспособная программа может содержать скрытые ошибки конкурентного доступа.

На этом фоне язык *Rust* представляет особый интерес, поскольку в нем организация безопасной работы с памятью и организация многопоточного взаимодействия строятся в рамках единой языковой модели [2]. Средства синхронизации в *Rust* согласованы с системой типов, механизмами владения и заимствования, что позволяет сократить класс ошибок, обнаруживаемых только на этапе выполнения [3].

Целью настоящей статьи является анализ средств синхронизации в многопоточном программировании и выявление особенностей их реализации в языке *Rust*.

Теоретические основы синхронизации в многопоточном программировании

Синхронизация в многопоточном программировании представляет собой совокупность методов согласования действий нескольких потоков, совместно использующих вычислительные ресурсы и данные. Ее основное назначение состоит в обеспечении корректного порядка выполнения операций и предотвращении конфликтов доступа к разделяемому состоянию.

Взаимодействие потоков обычно связано с тремя ключевыми задачами. Первая заключается в обеспечении взаимного исключения, когда изменяемый ресурс в каждый момент времени доступен только одному потоку. Вторая связана с координацией последовательности действий, когда один поток должен ожидать изменения состояния, формируемого другим потоком. Третья задача состоит в выполнении простых операций над данными атомарно, т. е. без промежуточных состояний, наблюдаемых другими потоками [1].

В зависимости от характера решаемой задачи в многопоточном программировании используются различные классы средств синхронизации. Наиболее распространенным примитивом является мьютекс, обеспечивающий эксклюзивный доступ к критической секции. Близким по назначению, но более гибким механизмом выступает блокировка чтения-записи, допускающая одновременную работу нескольких потоков-чтателей при отсутствии записи. Для ожидания определенного события или изменения состояния применяются условные переменные. Для построения высокопроизводительных и низкоуровневых структур используются атомарные операции. Наконец, важной альтернативой синхронизации на основе общей памяти служит модель обмена сообщениями между потоками.

Следовательно, сама проблема синхронизации в многопоточном программировании имеет не только прикладной, но и методологический характер: выбор механизма синхронизации влияет как на корректность программы, производительность, сопровождаемость и устойчивость к ошибкам.

Конкурентная модель языка Rust

Специфика *Rust* определяется тем, что многопоточное программирование рассматривается в нем не как отдельная библиотечная надстройка, а как продолжение общей модели безопасного доступа к данным. Основу этой модели составляют владение, заимствование и контроль времени жизни объектов [2].

Каждое значение в *Rust* имеет владельца, а правила передачи и временного использования значения строго формализованы. Одновременное существование нескольких изменяемых ссылок либо сочетание изменяемой и неизменяемых ссылок на одни и те же данные в обычном безопасном коде не допускается. Тем самым уже на уровне базовой семантики языка ограничиваются ситуации, потенциально ведущие к некорректному совместному доступу.

При переходе к многопоточному исполнению эта модель дополняется трейтами *Send* и *Sync*. Первый из них указывает, что значение может быть безопасно передано между потоками, второй — что значение может быть безопасно разделено между потоками через общую ссылку. Таким образом, язык вводит формальные ограничения

на использование типов в многопоточной среде [4; 5].

Принципиально важным является и то, что в Rust разделяемость и изменяемость не совпадают. Совместное владение объектом не означает допустимости его конкурентного изменения. Поэтому для организации общей изменяемой памяти необходимо явно использовать специальные средства синхронизации. В результате архитектура конкурентной программы выражается не неявно, а непосредственно в типах и структурах данных.

Такой подход отличается от традиционной модели, в которой библиотечные примитивы синхронизации нередко накладываются на язык, допускающий практически любой способ обращения к памяти. В *Rust*, напротив, сами допустимые способы совместного доступа ограничиваются заранее.

Основные средства синхронизации в Rust

Мьютекс

Mutex<T> является базовым средством взаимного исключения в *Rust*. Он предоставляет эксклюзивный доступ к значению типа *T*, защищая его от одновременного изменения несколькими потоками. После захвата блокировки поток получает специальный *guard*-объект, через который и осуществляется доступ к данным.

Существенным преимуществом такой организации является автоматическое освобождение блокировки при выходе *guard*-объекта из области видимости. Это снижает риск ошибок, связанных с ручным снятием блокировки, и делает управление жизненным циклом синхронизационного ресурса более надежным.

Практически мьютекс целесообразно использовать в ситуациях, когда критическая секция невелика, а операции чтения и записи приблизительно сопоставимы по частоте либо преобладает запись.

Блокировка чтения-записи

RwLock<T> предназначен для сценариев, в которых операции чтения выполняются значительно чаще операций изменения состояния. В отличие от мьютекса, данный примитив допускает одновременное существование нескольких читателей, если ни один поток не осуществляет запись.

Такой механизм позволяет повысить степень параллелизма и уменьшить число ненужных блокировок в системах с интенсивным чтением. Вместе с тем применение *RwLock* требует более внимательного отношения к характеру нагрузки, поскольку при высокой конкуренции могут возникать задержки писателей и иные эффекты, зависящие от политики планирования.

Условные переменные

Condvar используется для координации потоков в тех случаях, когда необходимо дождаться наступления определенного условия. Обычно условная переменная применяется совместно с мьютексом, защищающим состояние, проверка которого определяет продолжение или приостановку выполнения потока.

Содержательно *Condvar* не столько ограничивает доступ к данным, сколько организует ожидание их изменения. Это делает данный примитив необходимым в задачах построения очередей, пулов заданий и иных координационных механизмов.

Атомарные операции

Атомарные типы применяются для выполнения элементарных операций над данными без использования традиционных блокировок. Они позволяют реализовывать счетчики, флаги состояния, сигналы готовности и отдельные безблокировочные структуры данных. В *Rust* атомарные операции занимают важное место, поскольку именно на них, в частности, основан потокобезопасный подсчет ссылок.

Использование атомиков позволяет снизить накладные расходы на синхронизацию, однако требует более глубокого понимания модели памяти и порядка видимости операций. Поэтому данные механизмы являются более низкоуровневыми и обычно применяются там, где это действительно оправдано с точки зрения производительности.

Каналы передачи сообщений

Альтернативой модели общей памяти в *Rust* выступают каналы сообщений. Они позволяют организовать взаимодействие

потоков без непосредственного совместного доступа к изменяемым данным. Потоки обмениваются сообщениями, передавая владение значениями или их копии по согласованному протоколу.

Этот подход особенно хорошо сочетается с философией языка, основанной на явном владении. Во многих случаях каналы позволяют упростить архитектуру многопоточного приложения, поскольку уменьшают объем разделяемого состояния и тем самым снижают вероятность ошибок синхронизации.

Преимущества подхода Rust к синхронизации

Одним из главных преимуществ *Rust* является перенос значительной части ошибок из этапа выполнения на этап компиляции. Если тип не удовлетворяет требованиям безопасной передачи или совместного использования между потоками, компилятор запрещает соответствующую конструкцию. Это особенно важно для сложных многопоточных систем, где поиск дефектов времени выполнения требует значительных затрат.

Не менее важным преимуществом является явность архитектуры совместного доступа. Использование комбинаций *Arc<Mutex<T>>*, *Arc<RwLock<T>>* и иных аналогичных конструкций делает модель владения и синхронизации очевидной как для автора программы, так и для других разработчиков. Следовательно, код становится более прозрачным и легче поддается анализу.

Дополнительным достоинством выступает безопасное управление жизненным циклом блокировок. Автоматическое освобождение *guard*-объектов уменьшает риск зависаний, связанных с забытыми блокировками, и снижает влияние человеческого фактора.

Ограничения и практические аспекты

Несмотря на значительные преимущества, использование *Rust* в задачах многопоточности связано и с рядом ограничений. Прежде всего, строгая модель владения и заимствования требует от разработчика более высокой дисциплины проектирования и усложняет начальное освоение языка. Те решения, которые в иных языках записываются проще синтаксически, в *Rust* требуют более явного выражения.

Кроме того, *Rust* не устраняет автоматически все логические ошибки конкурентного программирования. Он позволяет эффективно предотвращать небезопасный доступ к памяти и большой класс ошибок совместного использования данных, однако не гарантирует отсутствия взаимных блокировок, *starvation*, неудачных протоколов обмена сообщениями и неэффективной декомпозиции задач.

Следовательно, *Rust* следует рассматривать не как универсальное средство полного устранения всех проблем многопоточного программирования, а как язык, существенно уменьшающий пространство опасных решений и повышающий общий уровень надежности программных систем.

Заключение

Проведенный анализ показывает, что средства синхронизации в языке *Rust* опираются на классические механизмы многопоточного программирования, но реализуются в рамках принципиально иной модели безопасной конкурентности. Мьютексы, блокировки чтения-записи, условные переменные, атомарные операции и каналы сообщений в *Rust* не существуют изолированно, а взаимодействуют с системой владения, заимствования и типовыми ограничениями.

Установлено, что данная архитектура позволяет существенно уменьшить число типичных ошибок конкурентного доступа, повысить надежность кода и сделать модель совместного использования данных более прозрачной [6]. Особое значение имеет возможность обнаружения значительной части нарушений уже на этапе компиляции.

Таким образом, язык *Rust* представляет собой перспективный инструмент разработки многопоточного программного обеспечения, в котором требования безопасности и производительности сочетаются более последовательно, чем в традиционных моделях системного программирования.

Список литературы

1. Herlihy M., Shavit N. The Art of Multiprocessor Programming. Revised reprint. Amsterdam: Morgan Kaufmann, 2012. 536 p.
2. Matsakis N. D., Klock II F. S. The Rust Language // Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology. New York: ACM, 2014. P. 103–104.
3. Klabnik S., Nichols C. The Rust Programming Language. 2nd ed. San Francisco: No Starch Press, 2023. 560 p.
4. Blandy J., Orendorff J., Tindall L. Programming Rust: Fast, Safe Systems Development. 2nd ed. Sebastopol: O'Reilly Media, 2021. 738 p.
5. Bos M. Rust Atomics and Locks: Low-Level Concurrency in Practice. Sebastopol: O'Reilly Media, 2023. 250 p.
6. Jung R., Jourdan J.-H., Krebbers R., Dreyer D. RustBelt: Securing the Foundations of the Rust Programming Language // Proceedings of the ACM on Programming Languages. 2018. Vol. 2. POPL. Art. 66. P. 1–34.