Санкт-Петербургский государственный университет

Максименко Дмитрий Сергеевич

Выпускная квалификационная работа

Реализация кластерного режима в SCSI target ядра Linux

Уровень образования: бакалавриат

Направление *02.03.03 "Математическое обеспечение и администрирование информационных систем"*

Основная образовательная программа *СВ.5006.2017 "Математическое обеспечение и администрирование информационных систем"*

Профиль *Системное программирование*

Научный руководитель:
доц. каф. СП к.т.н. Д. В. Луцив

Консультант:
Ведущий инженер-программист подсистем протоколов доступа в компании Yadro
К.А. Шелехин

Рецензент:
Ведущий инженер-программист в компании Yadro Р.О. Большаков

Санкт-Петербург
2021

Saint Petersburg State University

Maksimenko Dmitrii

Bachelor's Thesis

Implementing cluster mode in Linux kernel SCSI target

Education level: bachelor

Speciality *02.03.03 "Software and Administration of Information Systems"*

Programme *CB.5006.2017 "Software and Administration of Information Systems"*

Profile: *System Programming*

Scientific supervisor:
Sr. lecturer, C.Sc. Dmitrii Luciv

Consultant:
Leading Software Engineer of Subsystems of Access Protocols in Yadro Konstantin Shelekhin

Reviewer:
Principal Software Engineer at Yadro Roman Bolshakov

Saint Petersburg
2021

Оглавление

Введение

SCSI[1] (Small Computer System Interface) — это набор стандартов, описывающих механизмы, интерфейсы и команды для передачи данных между компьютерами и запоминающими устройствами. В ядре операционной системы Linux предусмотрена возможность выдавать доступ к запоминающим устройствам (как правило реальным или виртуальным дискам) по протоколам SCSI. Тогда сторона, которая владеет диском именуется SCSI-target, а сторона, которая читает или пишет в этот диск: SCSI-initiator. Initiator может посылать на target'ы команды согласно задокументированному протоколу, например, команду на получение единоличного доступа к диску. В ядре Linux присутствует подсистема, способная выполнять SCSI команды, в случае когда каждый диск управляется одним target'ом.

На практике зачастую возникает ситуация, когда доступ к одним и тем же дискам управляется несколькими SCSI-target серверами (Рис. 1). Такой подход используется для повышения отказоустойчивости системы и пиковой скорости чтения/записи и называется кластерным режимом. SCSI кластеры часто применяются для организации систем хранения данных. При таком подходе возникает проблема синхронизации состояний между SCSI-target серверами. На данный момент эта проблема решена различными способами в прикладных системах, но на уровне операционный системы с ядром Linux такой режим не поддерживается. В штатной подсистеме ядра Linux, TCM (Target Core Mod)[1][2], которая включает SCSI-target функциональность, нет поддержки кластерного режима SCSI-target. В данной работе планируется разработать метод, с помощью которого можно реализовать кластерный режим в данной подсистеме.

Заметим, что если подключить несколько SCSI-target'ов без поддержки кластерного режима к одним и тем же дискам, система будет работать некорректно в некоторых случаях. Например, если клиент 1 через target 1 начнет взаимодействовать с диском 1 (например, писать

[1]Также именуется LinuxIO или LIO.

на него данные), при этом клиент 2 через target 2 также начнет писать на диск 1 данные, то данные на диске 1 окажутся в некорректном состоянии.

Стоит отметить, что наличие кластерного режима в SCSI-target ядра Linux еще не означает, что существует возможность создавать эффективные кластеры средствами операционной системы. При работе кластера необходимо правильно его конфигурировать, обрабатывать присоединения и отсоединения узлов во время работы кластера, регулировать работу узлов в случае распада кластера, когда часть узлов теряет соединение с остальными узлами, и перезапускать узлы по мере необходимости. Всё это тяжелые задачи, для которых существуют готовые решения в виде программ пользовательского пространства. Данная работа предполагает использование одного из таких решений.

В процессе изучения команд и механизмов SCSI был выделен механизм постоянных резерваций (Persistant Reservations), как наиболее важный механизм для создания систем хранения данных, построенных на SCSI кластерах, поэтому в данной работе ему уделено особое внимание. Этот механизм оперирует резервациями, сущностями, описывающими взятие некоторого ресурса, например диска, под контроль клиентом или группой клиентов. В стандарте также описаны команды, применяемые при работе с резервациями. Резервации могут быть постоянными, что означает, что они сохраняются при сбоях на аппаратном или сетевом уровнях, например, при сбое питания на клиенте, владеющем резервацией, или при аварийном отказе на SCSI-target'е. Этот механизм является примером функциональности, реализация которой в кластерном режиме требует дополнительных взаимодействий в кластере, например, нам нужно синхронизировать информацию о резервациях между всеми SCSI-target'ами.

Существуют и другие команды SCSI, реализация которых в кластерном режиме требует синхронизации данных. Так в данной работе была реализована команда «Compare and Write», аналог команды «CMPXCHG» (Compare and Exchange)[3] процессоров с архитектурой x86. Таким образом, команда «Compare and Write» может служить при-

митивом для создания семафоров на уровне кластера.

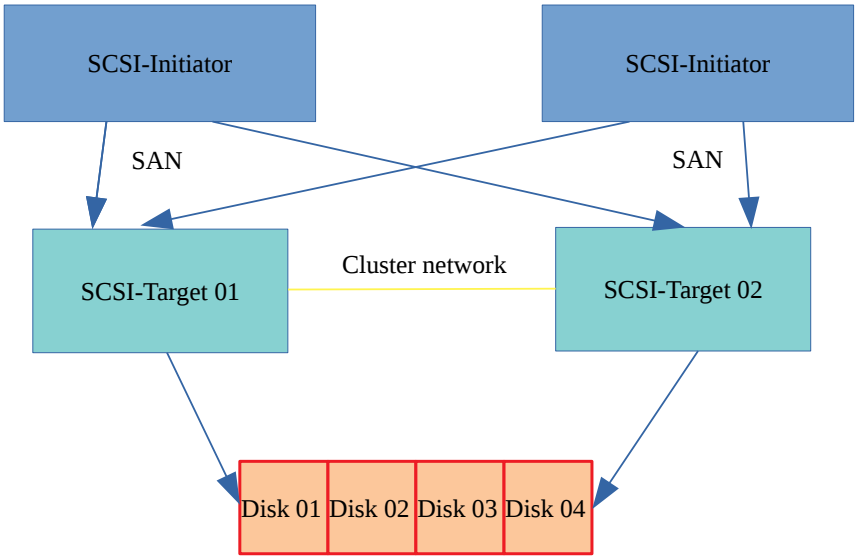

Рис. 1: Кластерный режим

1. Постановка задачи

Данная работа посвящена разработке метода реализации кластерного режима в SCSI-target подсистеме ядра Linux, TCM, что позволит добавлять в команды SCSI кластерный режим, необходимый для работы системы, в которой доступ к запоминающим устройствам контролируется несколькими SCSI-target'ами. Для достижения цели были сформулированы следующие задачи:

- провести исследование предметной области, изучить существующие стандарты;
- произвести обзор решений, которые позволяют организовать SCSI кластер и зафиксировать лучшие аспекты этих решений;
- разработать метод реализации кластерного механизма в SCSI-target ядра Linux;
- реализовать разработанный метод и добавить с помощью него кластерный режим в некоторые команды SCSI;
- провести тестирование ключевой функциональности с помощью тестовых стендов.

Полученные результаты планируется передать компании Yadro для дальнейшей доработки с последующим встраиванием данной функциональности в открытую версию ядра Linux.

2. Исследование предметной области

2.1. SCSI кластеризация

Существенной частью данной работы было изучение предметной области: существующих технологий и стандартов. Для лучшего знакомства с технологиями был собран тестовый стенд, состоявший из двух виртуальных машин, взаимодействующих по протоколу iSCSI (SCSI поверх IP).

Одна из машин представляла SCSI-target, предоставлявший доступ к своему виртуальному диску по протоколу iSCSI. Для настройки работы таргета использовалась утилита targetcli[4]. В качестве самого SCSI-target'а использовалась встроенная в ядро Linux подсистема TCM.

Вторая машина представляла SCSI-initiator, который получал доступ к диску на первой машине. Для работы инициатора использовалась утилита open-iscsi[5].

Виртуальные машины запускались с помощью программы VirtualBox[6] и работали на Ubuntu 20.04.

В качестве теоретической подготовки были изучены стадарты SCSI.

С помощью стандартов SCSI Primary Commands-4[7] и SCSI Block Commands – 3[8] были изучены команды SCSI. Особое внимание уделялось командам, изменяющим или задающим состояние системы и командам механизма Persistent Reservations.

Для того, чтобы понять как в SCSI организовано клиент-серверное взаимодействие и какие основные логические сущности существуют в SCSI, был изучен стандарт SCSI Architecture Model-5[9].

2.2. Persistent Reservations

Важной частью изучения предметной области стало изучение механизма Persistent Reservations — ключевого механизма, необходимого для построения кластеров. Данный механизм используется для того, чтобы клиенты могли брать блокировки на ресурсы, тем самым предотвращая работу с этим ресурсом другого клиента. В качестве таких ре-

сурсов обычно выступают LUN'ы (Logical Unit Number), логические номера, соответствующие запоминающим устройствам или их разделам.

В механизме Persistent Reservations существуют всего две команды, управляющие доступом к устройствам: PERSISTENT RESERVE OUT (PR OUT) и PERSISTENT RESERVE IN (PR IN). При этом, эти две команды могут вызывать много различных действий на сервере, в зависимости от аргументов.

Существует такая сущность, как путь между SCSI-initiator и SCSI-target. Между одной парой инициатора и таргета может быть несколько путей. Зачастую путь соответствует физическому проводу между двумя компьютерами. В системах хранения данных инициаторы и таргеты бывают соединены десятками таких проводов.

Для начала работы требуется зарегистрировать ключ резервации. На каждый путь может быть зарегистрировано не более одного ключа. Затем с помощью ключа резервации клиент можем сопоставить некоторый идентификатор каждому пути. С помощью команды PR IN можно узнавать, какому пути принадлежит резервация конкретного ресурса. У клиента есть возможность регистрировать путь с любой комбинацией логических устройств таргета, что позволяет в последствии резервировать их одной командой.

Команда PR IN предназначена для получения информации клиентом и может вызывать такие действия на сервере: READ RESERVATION, READ KEYS, READ FULL STATUS.

Команда PR OUT предназначена для вызова действий на сервере, которые меняют состояние резерваций. Два действия разрешены с использованием незарегистрированного ключа: зарегистрировать ключ (REGISTER) и зарегистрировать ключ, игнорируя существующий (REGISTER AND IGNORE EXISTING KEY). Остальные действия требуют зарегистрированного ключа (например, RESERVE - зарезервировать, RELEASE - снять резервацию и тп).

Таким образом, клиент посылает команды и ждет на них ответ от сервера. На сервере или в кластере из серверов действие резервации должно быть атомарным: резервация либо полностью произошла, либо

не произошла никак.

Как было сказано выше команда PR OUT может вызывать действие на сервере, которое может изменить состояние резерваций. Некоторые команды могут быть отклонены сервером: команды с некорректными аргументами или команды, которые не могут быть исполнены в данный момент (например, попытка взять резервацию, которая уже взята). Остальные же команды исполняются и переводят систему в другое состояние[2] и от этих состояний зависит поведение системы при исполнении последующих команд. А поскольку в кластер могут поступать несколько команд одновременно, то нужно продумать поведение системы и на этот случай. Для этого в стандарте SCSI присутствуют конечные автоматы, которые описывают поведение системы при поступлении любой команды из стандарта для любого допустимого состояния. Удобно демонстрировать автоматы на примере диаграмм, так на изображении (рис. 4) продемонстрировано поведение системы в случае запроса действия PREEMPT в команде PR OUT. Действие PREEMPT служит для передачи резервации от одного пути к другому. Его ключевые аргументы это ключ резервации (id пути которому принадлежала резервация), ключ резервации действия сервера (id пути которому передается резервация), новый тип блокировки и новая область видимости блокировки. Также стоит отметить, что значение ключа ноль является специальным и означает, что ключ не действителен.

Исходя из этого исследования были выделены основные сущности механизма и взаимосвязь между ними, что в последствии использовалось при продумывании взаимодействия между SCSI-target и модулем ядра, отвечающим за синхронизацию.

[2]В данном контексте под словом состояние подразумевается не только множество дисков и ограничений на доступ к ним, но и внутреннее состояние системы, включающее, например, множество зарегистрированных ключей.

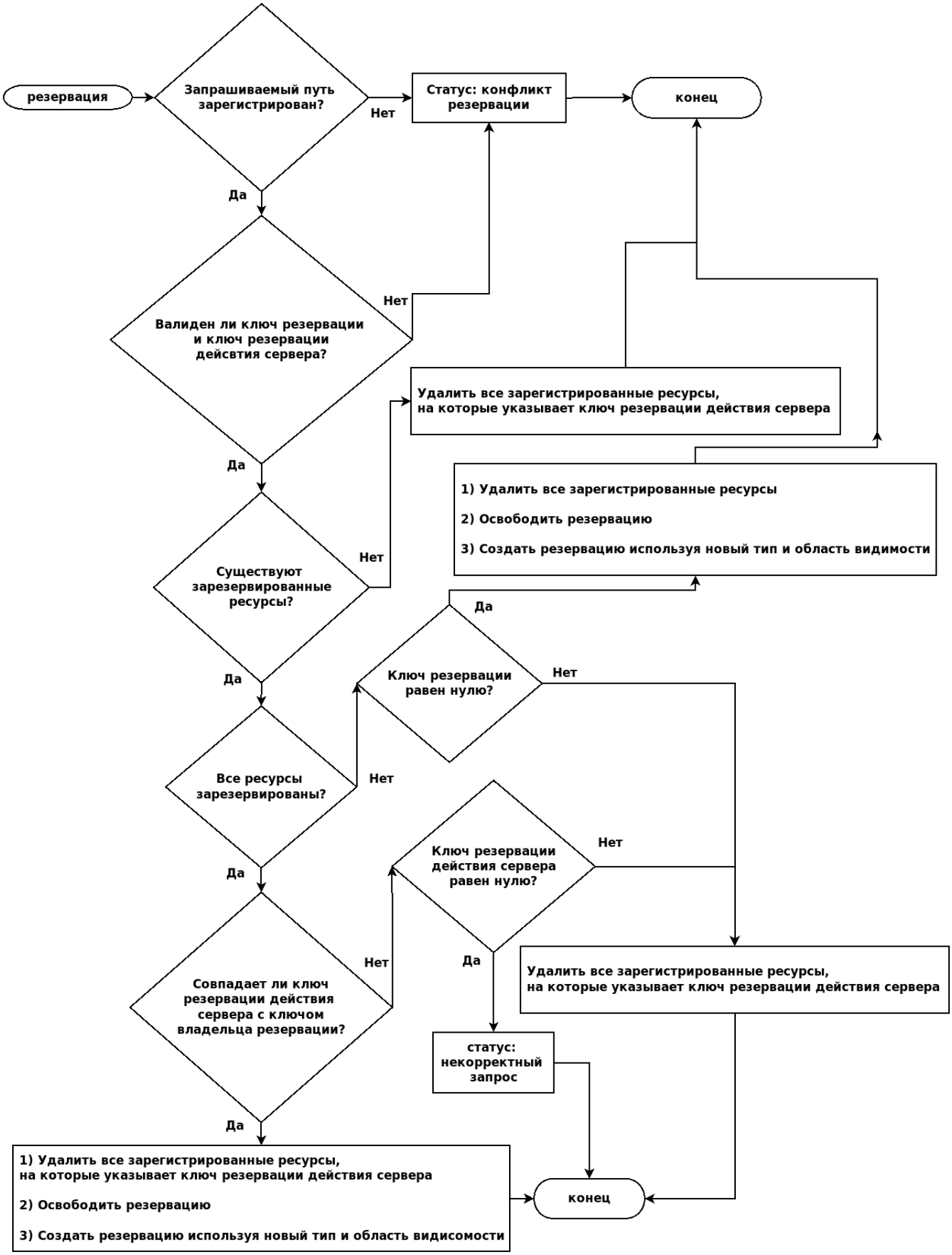

Рис. 2: Алгоритм обработки действия PREEMPT

3. Обзор существующих аналогов

Существует несколько работающих реализаций кластерного режима SCSI-target. Данная работа во многом опирается на опыт предыдущих подходов. Её главным преимуществом будет нацеленность на встраивание функциональности в непосредственно ядро Linux. Рассмотрим несколько успешных реализаций от производителей, которые по тем или иным причинам не могут быть встроены в ядро Linux.

Одной из таких реализаций является SCST[10] от компании SanDisk. SCST предоставляет аналог присутствующей в ядре подсистемы TCM. До 2011 года TCM и SCST конкурировали за возможность быть встроенными в ядро по причине того, что я ядре может быть только одна SCSI-target подсистема. В конечном итоге выбор пал именно на TCM[11].

SCST предоставляет возможность создавать репликации, работать с тонкими ресурсами и производить резервное копирование. Работает с такими типами транспортных протоколов, как: iSCSI, Fibre Channel, FCoE, SAS, InfiniBand (SRP) и Wide (parallel) SCSI.

Ключевой частью SCST является ядро SCST, которое встраивается между драйверами SCSI и запоминающими устройствами, производит предварительную обработку поступающих SCSI запросов и обработку ошибок. Позволяет настраивать видимость SCSI таргетов SCSI клиентами. Позволяет брать блокировки на ресурсы. Поддерживает многопоточный режим, при котором скорость обработки команд ускоряется в количество раз, равное количеству процессоров. Утверждается, что SCST незначительно увеличивает вычислительную сложность обработки команд. На данный момент SCST является лидером рынка по количеству поддерживаемой функциональности[12].

Данная реализация опирается на механизм менеджера распределенных блокировок (DLM)[13], который был добавлен в ядро Linux в качестве драйвера ядра в 2006 году. Механизм постоянных резерваций (являющийся частью стандарта SCSI) был реализован с помощью создания структур, хранящих всю информацию о состоянии резерваций

(такую как количество зарегистрированных объектов, тип резервации, пространство резервации), и синхронизацией этих структур между узлами кластера с помощью DLM.

SCSI-target узлы узнают об изменениях в резервациях через уведомления. Каждый узел содержит полную копию конфигурации и может восстанавливаться из неё в случае сбоя питания.

SCST может решать задачу кластеризации, но имеет существенные недостатки. Данное решение не расширяет существующую в ядре подсистему, а создает новую. Кроме того, при создании данного проекта его руководство не консультировалось с представителями ядра, в следствие чего многие аспекты работы SCST, например, интерфейсы, оказались неудобными к интеграции с другими частями ядра. По мнению людей, ответственных за открытую версию кода Linux, такой подход не соответствует принципам развития ядра. Решение, разрабатываемое в рамках данной работы будет во многом опираться на опыт SCST, но не будет обладать этими недостатками.

Еще один подход представлен в подсистеме CTL[14], разработанной компанией Copan Systems в 2003 году и портированной на ОС FreeBSD из Linux в 2008. CTL включает SCSI-target подсистему. Среди прочего CTL поддерживает механизм постоянных резерваций и выполнение SCSI команд в заданном порядке. CTL представляет собой драйвер, который может быть скомпилирован или загружен в ядро FreeBSD. CTL предоставляет возможность создавать кластеры (High Availability cluster) с помощью механизма CTL HA, но возможности создавать классические SCSI кластеры нет. Данное решение безусловно интересно, так как позволяет создавать кластеры средствами операционной системы. В рамках данной работы также планируется разработать метод для реализации возможности создавать кластеры средствами операционной системы (за исключением кластерного ПО), но в отличии от CTL, данная возможность появится в более популярных операционных системах использующих ядро Linux (например Ubuntu), а также с использованием общепринятого протокола SCSI.

4. Разработка метода

4.1. Выбор средства синхронизации

Как было установлено ранее, синхронизация данных является ключевым моментом в реализации кластерного режима, поэтому нам необходимо какое-то средство, которое будет предоставлять возможность синхронизировать данные между узлами кластера (SCSI-target'ами). Данное средство должно удовлетворять следующим требованиям: оно должно присутствовать в ядре Linux, предоставлять возможность передавать данные между любыми двумя узлами в системе и желательно корректно работать в конкурентной многопоточной среде. Последнее желательно, поскольку SCSI команды могут поступить на различные узлы одновременно, что повлечет попытки одновременного распространения информации, что в свою очередь может привести к различным гонкам и взаимоблокировкам.

В ядре Linux достаточно средств, которые позволяют возможность передавать информацию по сети. Так, например, можно использовать tcp/ip сокеты. Но в такого рода средствах отсутствуют инструменты поддержки «кластерности», все что они позволяют — это передавать сообщения, что означает, что придется реализовать очень многие вещи, такие как методы предотвращения взаимоблокировок, очереди запросов и прочее.

Существуют и противоположные по сложности решения: например, кластерные файловые системы, такие как OCFS2[15]. С помощью них можно иметь синхронизированные в кластере файлы, что является примитивом для передачи информации. Например, узел, получивший команду, может писать её в файл по определенным правилам, а другие узлы будут читать её и исполнять локально. Кроме того, данный способ практически избавляет нас от необходимости переживать за некорректное состояние этого файла, так как кластерная файловая система берет многое на себя. Недостатком данного способа является его тяжеловесность, что может привести к низкой производительности системы.

Кроме того, с помощью данного инструмента крайне сложно избавиться от активного ожидания: необходимости читать синхронизированный файл, ожидая пока там появится информация.

Правильным решением кажется использовать инструмент, который как раз и делает такие файловые системы, как OCFS2 кластерными. Этот инструмент называется Distributed Lock Manager (DLM). Он входит в ядро Linux с 2006 года. DLM предоставляет такие сущности, как «пространства блокировок», которые в свою очередь могут содержать «блокировки», ключевые сущности в работе DLM. Узлы в кластере могут создавать эти блокировки, а также блокировки содержат специальное значение, которое можно синхронизировать в кластере. Размер этого значения можно конфигурировать. В данной работе размер было выставлен в 256 байт, поскольку это было достаточно для того, чтобы помещать необходимую информацию в это значение. DLM реализует шесть режимов блокировки: эксклюзивный доступ, защищенная запись, защищенное чтение, одновременная запись, параллельное чтение и отсутствие блокировки. Кроме того, в DLM существуют очереди запросов на взятие блокировок, которые не могут быть удовлетворены сразу: одна очередь для созданния блокировок, а другая для преобразования блокировок к новому режиму. Также присутствует механизм детектирования взаимоблокировок, которые могут возникнуть в этих очередях. Кроме этого в DLM есть возможность регулировать поведение блокировок в случае сбоев питания, что сыграет ключевую роль в реализации механизма SCSI Persistent Reservations. Все это делает DLM идеальным инструментом для реализации поставленных целей, кроме того есть пример успешной реализации похожей функциональности с использованием DLM: SCST.

Также для функционирования кластера требуется кластерное программное обеспечение, которое будет контролировать состав кластера, перезапускать узлы кластера и создавать кластерные ресурсы. В качестве такого инструмента был выбран стек Pacemaker/Corosync[16], как самое распространенное решение среди аналогов. Модуль синхронизации может быть в достаточной степени абстрагирован от Pacemaker'a

и при небольшой модификации работать и с его аналогами.

4.2. Активные или пассивные обновления

При построение распределенной системы знаний, которой является набор состояний системы на всех узлах кластера, существует требование, чтобы запрашиваемая информация была актуальна. Существует два подхода к выполнению этого требования.

Первый подход состоит в том, чтобы при запросе информации на каком-то узле, этот узел осуществлял запрос информации и с других узлов и возвращал самую актуальную информацию. При этом в нашем случае передавать между узлами можно не все состояние, а только изменения, произошедшие с предыдущего запроса.

Второй подход заключается в распространении изменений по всем узлам при поступлении этого изменения.

Оба подхода имеют свои преимущества и недостатки. Так первый подход существенно проще в реализации. Команды записи состояния (изменяющие состояние системы) не используют кластерную передачу данных, но при этом команды чтения состояния требуют кластерные взаимодействия в количестве равном количеству узлов в кластере (так как нужно запросить информацию со всех остальных узлов).

Второй подход лишен этого недостатка, так как информация на каждом узле находится в актуальном состоянии, но команды записи состояния начинают требовать кластерные взаимодействия, опять же в количестве равном количеству узлов в кластере (нужно распространить изменения на все остальные узлы). Второй подход также требует некоторых средств для его реализации, например, чтобы была возможность оповестить узел об обновлении (не требуя, чтобы оповещаемый узел находился в активном ожидании).

Так что выбор между этими подходами сводится к ожидаемому соотношению команд записи и чтения состояния и наличию средств, необходимых для второго подхода. И поскольку в системах хранения данных, использующих SCSI кластеры, операции чтения состояния, как

правило происходят на порядки чаще, и в DLM есть необходимые для реализации второго подхода инструменты, был выбран второй подход.

Поясним, почему в системах хранения данных, использующих SCSI кластеры, операции чтения состояния, как правило, происходят на порядки чаще чем команды записи состояния.

Вспомним, что командами чтения состояния мы называем те, которые не изменяют состояние системы, поэтому команда записи (WRITE) в данном контексте тоже является командой чтения состояния. Перед началом исполнения команды записи, проверяется наличие резерваций. Для этого необходимо прочитать состояние ключей резервации и проверить что данный инициатор имеет право на доступ к ресурсу (LUN'у).

Командами записи состояния являются, например, команды взятия доступа к дискам, регистрации ключей в системе и так далее. Программное обеспечение, которое играет роль SCSI-initiator'ов, получая запросы от клиентов и транслируя их в последовательность действий непосредственно с системой хранения данных, выстраивает план запроса, запрашивая состояние системы (операция чтения состояние). Затем берет доступ к необходимым дискам (операции записи состояния) и в течение некоторого времени (речь может идти про секунды и минуты) работает в этими дисками (операции чтения состояния). Таким образом, операции записи состояния действительно составляют меньшинство.

4.3. Архитектура системы

В первом приближении была выбрана слоистая архитектура (рис. 3), состоящая из следующих слоев: сначала идет слой SCSI-target, который непосредственно исполняет команды SCSI. Этот слой представлен системой TCM, которая была доработана обращениями к следующему слою. Для реализации механизма Persistent Reservations также потребуется добавить в неё дополнительные структуры и их использование. В архитектуре TCM можно выделить основной модуль «target_core_mod», а также дополнительные модули: «target_core_file»,

«target_core_user» и «target_core_iblock», которые используются основным модулем по мере необходимости.

Следующий слой — это синхронизирующий модуль, ключевая часть данной работы. Синхронизирующий модуль отвечает за синхронизацию данных в кластере и предоставляет простой интерфейс для TCM. Он будет является еще одним дополнительным модулем для «target_core_mod» аналогично модулям, перечисленным ранее, и будет отвечать за кластерный режим.

Также подслоем синхронизирующего модуля можно выделить слой взаимодействия с синхронизирующими технологиями: DLM и Pacemaker. Наличие такого подслоя позволит при необходимости заменить DLM и Pacemaker на другие технологии. При этом замена Pacemaker'а на аналог почти не потребует изменений в коде. Замена же DLM на другое средство синхронизации невозможна в синхронизирующем модуле, созданном в рамках данной работы. Но наличие данного слоя все равно позволит легче переключаться на другие версии DLM.

Таким образом, предлагаемое решение является расширением штатной подсистемы ядра Linux: TCM, которая основана на движке SCSI и согласуется с архитектурой, описанной в SCSI Architecture Model.

Рассмотрим также возможную взаимную сборку SCSI-target и синхронизирующего модуля. Возможны три случая: они компилируются совместно, компилируются отдельно и линкуются совместно или компилируются и линкуются отдельно. В рамках этой работы был выбран третий способ, поскольку он позволяет сделать SCSI-target и синхронизирующий модуль независимыми, что облегчит переиспользование модуля в других подсистемах, таких как NVMeoF-target (nvmet)[17], где присутствует схожий со SCSI Persistent Reservations механизм синхронизации. Для того, чтобы SCSI-target мог обращаться к интерфейсу синхронизирующего модуля, синхронизирующий модуль экспортирует необходимые функции. А SCSI-target собирается с использованием экспортируемых символов синхронизирующего модуля.

Поведение системы при поступлении команд чтения не отличается от изначального, поскольку они не изменяют состояния на узле, а

значит не требуют синхронизации.

Поведение системы при поступлении команд записи можно описать так: TCM применяет изменение на локальном узле, после чего с помощью синхронизирующего модуля распространяет обновление на остальные узлы. Если распространение совершено успешно,
SCSI-initiator'у возвращается ответ о том, что команда применена успешна. Если же при распространении обновления произошли ошибки (например обновление не может быть применено), то обновление на локальном узле откатывается и SCSI-initiator'у возвращается ответ о том, что команда не применена с кодом ошибки.

Кроме того, стоит отметить, что в SCSI некоторые команды должны быть атомарны. В данной работе слово «атомарность» используется в том смысле, в котором оно используется при описании транзакций баз данный, то есть атомарная команда должна представлять непрерывную последовательность действий и либо выполняться полностью, либо никак. При этом атомарная команда не обязана быть атомарной с точки зрения работы процессора. Для осуществления этого требования синхронизирующий модуль предоставляет возможность блокировки ресурсов. Таким образом, общий принцип выполнения атомарных команд будет: взять блокировки на все затрагиваемые ресурсы, совершить изменения, снять блокировки.

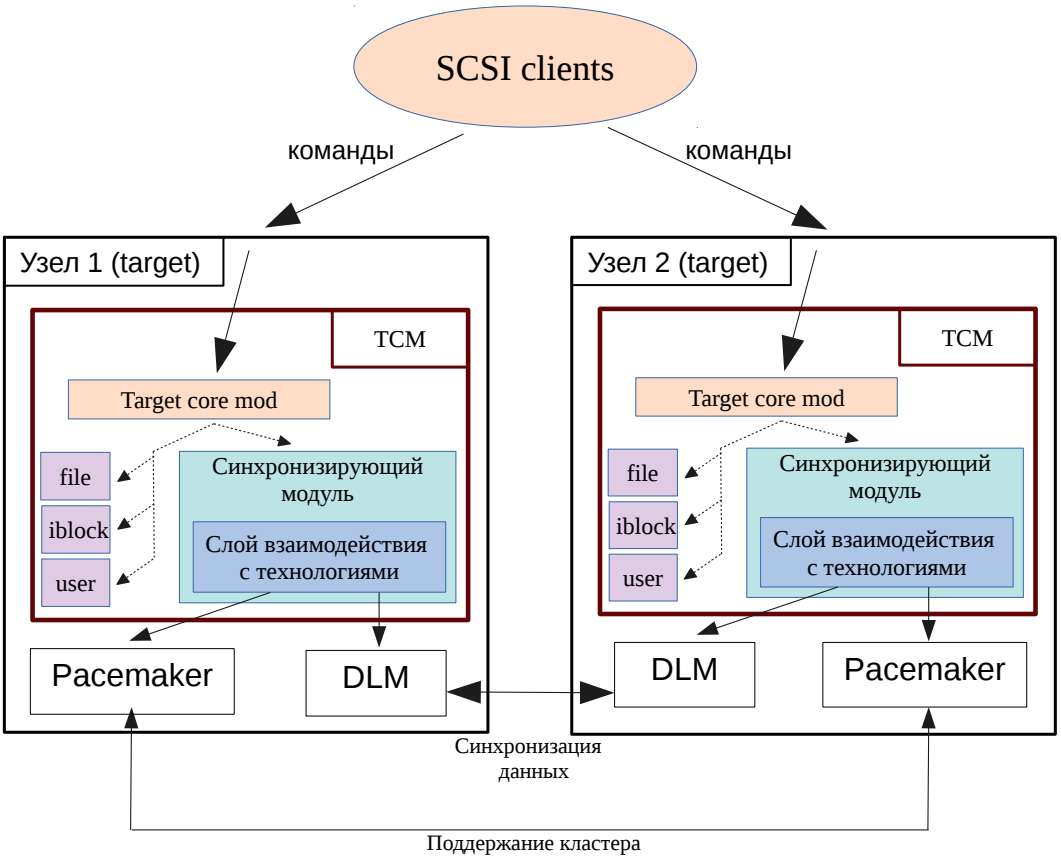

Рис. 3: Архитектура

5. Реализация

5.1. Синхронизирующий модуль

Ключевой компонентой данной работы является синхронизирующий модуль. Задача этого модуля обеспечивать синхронизацию информации в кластере и предоставлять простой интерфейс для его использования. Было решено, что в качестве средства синхронизации будет использован DLM, и что при поступлении обновления узел будет оповещать об этом остальные узлы кластера.

Кроме того, в связи с тем, что в DLM действия с блокировками выполняются по имени или id блокировки, было решено сделать интерфейс, состоящий из использования информации по схеме ключ-значение. Основными функциями стали: создание нового ключа со значением, эта же функция используется для установки нового значения ключу, взя-

тие блокировки на ключ и снятие блокировки.

Для того, чтобы продумать алгоритм оповещения об изменениях, нужно разобраться в том, как именно с помощью DLM можно распространять информацию. В DLM есть блокировки, которые содержат значение, которое называется Lock Value Block (LVB)[18]. Это значение синхронизируется в системе автоматически следующим образом: если блокировка, которая соответствует данному LVB конвертируется к более строгому режиму, значение LVB возвращается, а если к более слабому, то значение пишется[3] .

Кроме того в DLM есть два механизма обратных вызовов (callback'ов)[19][20]. При создании или конвертации блокировки ей можно присвоить данные обратные вызовы. Один из обратных вызовов называется AST (Asynchronous System Trap) и он вызывается, когда запрос, связанный с этой блокировкой, удовлетворяется. Второй называется BAST (Blocking AST) и вызывается на узле, являющимся держателем блокировки, которая не позволяет удовлетворить текущий запрос. Например, если была создана новая блокировка и в качестве AST была указана функция func1, а в качестве BAST — func2, то после того как блокировка создастся в системе, на узле, инициировавшем запрос, будет вызвана func1. Если в последствии будет сделан относящийся к этой же блокировке запрос, который не может быть осуществлен совместно с изначальной блокировкой, то будет вызвана func2 на узле, который создал начальную блокировку.

Запросы на создание или конвертацию блокировок в DLM являются асинхронными, но некоторые функции, которые предоставляет интерфейс данного модуля, должны быть синхронными, то есть возвращать управления после того, как проведена вся работа с блокировками DLM. Например, функция интерфейса синхронизирующего модуля «get_lock» должна возвращать управление только после того, как взята блокировка. Для этого в данной работе с помощью механизма AST и механизма ядра completion[21] был реализован синхронный DLM запрос. Для этого перед запросом инициализируется специальный объект,

[3]Это означает, что LVB примет это значение на всех узлах кластера.

который имеет состояние «не завершен», а после запроса ожидается пока этот объект не перейдет в состояние «завершен». Смена состояния происходит в AST, соответствующем этому запросу. Таким образом поток управления ожидает, пока асинхронный запрос на блокировку не завершится.

Кроме того, стоит отметить, что механизмы AST и BAST не должны быть блокирующими: так спроектирована логика DLM, поэтому если внутри некоторых обратных вызовов требовалось сделать DLM запрос, то это выносилось в отдельную отложенную «работу» с помощью механизма ядра workqueue[22], который как раз предоставляет сущность «работа» и позволяет выносить работы в отдельные потоки.

Теперь обсудим основной алгоритм, который передает обновление с узла-обновителя на обновляемый узел. Для осуществления работы этого алгоритма в системе изначально создается по две DLM блокировки на каждый узел: назовем их post и pre. Тот узел, которому соответствуют эти блокировки берет pre с режимом EX (exclusive - самая сильная блокировка), post в режим NL (null - самый слабый режим, фактически отсутствие блокировки). Все остальные узлы берут эти блокировки в режим (NL). Отметим, что одна и та же блокировка может быть взята двумя узлами в режимы EX и NL, но не может быть взята двумя узлами в режим EX. К pre блокировке также выставляется BAST функция: pre_bast. Это начальное состояние, которое должно совпасть с состоянием после завершения обновления, чтобы можно было передать следующее обновление.

Обновляющий алгоритм (пусть для простоты обновляющий узел имеет номер 1, обновляемый 2, тогда блокировки pre_2, post_2, через которые осуществляется передача обновления, будем называть просто pre, post):

1) Узел 1 запрашивает блокировку pre в режим EX (на рисунке 4 обозначено, как «запрос 1»). Поскольку pre принадлежит узлу 2 с режимом EX, то запрос кладется в очередь (два режима EX не совместимы), а на узле 2 вызывается pre_bast, поскольку блокировка на

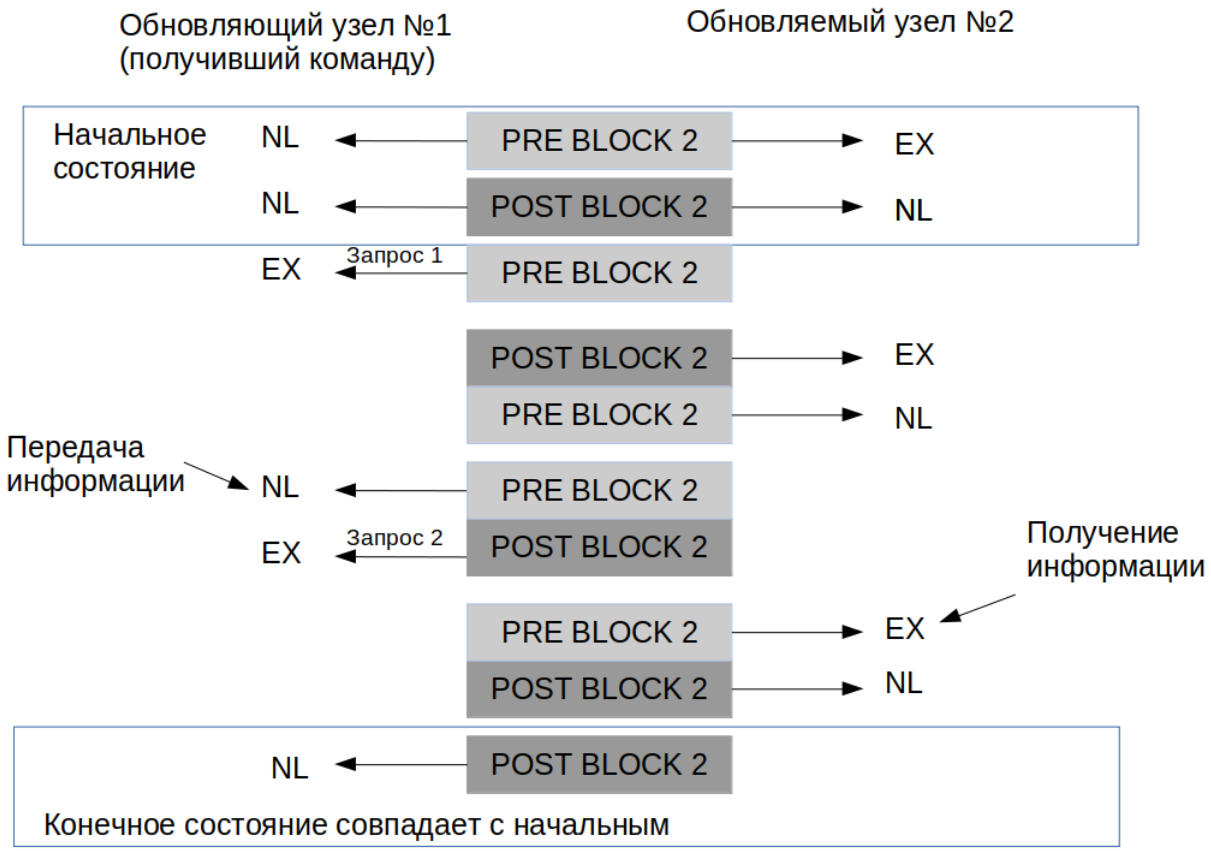

Рис. 4: Алгоритм распространения обновлений

узле 2 является блокирующей .

2) pre_bast на узле 2 переводит post в режим EX указывая в качестве BAST post_bast. Блокировку pre переводит в режим NL.

3) Теперь запрос из пункта 1 удовлетворен (поскольку режимы EX и NL совместимы), поэтому на узле 1 вызывается AST (так как «запрос 1» удовлетворён), который просто переводит объект completion в режим завершен, тем самым продолжая выполнение алгоритма на узле 1. Узел 1 пишет в LVB блокировки pre значение обновления[4].

4) Узел 1 конвертирует pre в режим NL и тем самым пишет новое значение LVB в систему (так как произошла конвертация к более слабому режиму).

5) Узел 1 запрашивает post в режим EX (на рисунке 4 обозначено, как «запрос 2»). Поскольку post принадлежит узлу 2 с режимом EX, то запрос кладется в очередь (два режима EX не совместимы), а на узле

[4]На самом деле в реализации используется еще одна блокировка — для передачи значений, но концептуально это не изменяет алгоритм, поэтому здесь эта деталь опущена.

2 вызывается post_bast.

6) post_bast на узле 2 переводит блокировку pre в режим EX, тем самым значение обновления читается из системы (так как произошла конвертация к более строгому режиму). Применяет это обновление.

7) post_bast на узле 2 переводит POST в режим NL.

8) Теперь запрос из пункта 5 удовлетворен и выполнение алгоритма продолжается на узле 1 (аналогично пунтку 3). Узел 1 переводит post в режим NL.

Алгоритм завершен, система снова находится в начальном состоянии. Обновление передано с узла 1 на узел 2.

Значение LVB также используется для передачи с обновляемого узла кода ошибки, если такая возникнет. Обновляющий узел проверяет это значение на шагах 3 и 5, и если передан код ошибки, то обновление откатывается.

Обновление представляет из себя структуру, для которой реализована возможность определенным способом заполнять буфер. Эта структура содержит поля, которые могут передавать всю необходимую для синхронизации SCSI состояний информацию.

Также синхронизирующий модуль может с помощью слоя взаимодействия с Pacemaker'ом получать информацию об узлах в кластере и имени текущего узла и использовать эту информацию при инициализации.

5.2. Драйвер символьного устройства

Для облегчения ручного и автоматического тестирования синхронизирующего модуля был разработан специальный драйвер символьного устройства[23]. Символьное устройство — это вид устройства в Linux-системах, предоставляющее возможность потокового символьного чтения и записи данных.

Для драйвера символьного устройства можно определить функции инициализации, чтения и записи. При этом операции чтения и записи

работают с клиентским буфером. Для передачи данных превышающих в размере размер буфера используется специальный параметр, отвечающий за смещение.

В рамках данной работы был реализован синхронизирующий модуль следующим образом. Операция инициализации символьного устройства запускает также инициализацию синхронизирующего модуля.

Операция чтения возвращает в символьном виде всю информацию про состояние системы в виде набора пар: ключ и значение и набора заблокированных ресурсов в виде пар: ресурс - владелец блокировки (данная блокировка не соответствует ни блокировкам DLM, ни резевациям SCSI Persistent Reservations, а является внутренней блокировкой, нужной в основном для реализации синхронизирующих команд).

Операция записи обрабатывает переданные символы и интерпретирует их, как команду из доступного набора команд и список её аргументов. Если команда введена правильно, вызывает соответствующую команду из интерфейса синхронизирующего модуля.

Таким образом, с помощью данного символьного устройства программы пользователей смогут использовать синхронизирующий модуль с помощью потокового ввода и вывода, а поскольку синхронизирующий модуль может синхронизировать любой набор данных, представимый в виде ключ - значение, то эта часть данной работы может быть переиспользована в других проектах, где требуется кластерная синхронизация.

5.3. Представление данных Persistent Reservations

Исходя из разработанной архитектуры, синхронизирующий модуль ничего не знает о данных, которые он синхронизирует, поэтому ответственность за то какие данные и когда синхронизировать остается в руках SCSI-target.

Особенного подхода требует механизм Persistent Reservations, потому что в отличии от большинства команд SCSI, его команды работают с внутренним состоянием: набором резерваций, участников, их ключей

и так далее. Достаточно интересной кажется идея отождествить резервацию механизма Persistent Reservations с блокировкой, которую нам предоставляет синхронизирующий модуль. Это могло бы быть осуществимо в старой версии Persistent Reservations, но в актуальной версии у резервации может быть сразу несколько владельцев (узлов кластера), а разработанный синхронизирующий модуль не позволяет этого сделать, поэтому информация о резервациях будет храниться в специальных структурах в TCM, которые будут синхронизироваться посредством синхронизирующего модуля.

Для каждого Logic Unit Number (LUN), то есть сущности запоминающего устройства, SCSI-target просит синхронизирующий модуль создать отдельное пространство блокировок. Это не нарушает логику системы, так как взаимодействие с одним LUN'ом никак не влияет на другие LUN'ы. Это положительно для системы, так как синхронизирующий модуль не может распространять более одного обновления в одном пространстве блокировок одновременно. Таким образом, при выделении пространства блокировок на каждый LUN увеличивается максимальная скорость передачи обновлений в системе.

Затем для каждого LUN создается одна специальная структура, которая содержит: количество зарегистрированных участников, тип резерваций, пространство резервации и два специфичных значения: является ли резервация персистентной и значение, характеризующие поведение при сбое питания. Для синхронизирующего модуля все эти данные хранятся, как значение, а ключом является специальное значение. Использовать специальное значение в качестве ключа возможно, поскольку данная структура уникальна для пространства блокировок.

Кроме того, для каждого LUN'а SCSI-target хранит структуры с информацией об участниках, содержащие: ключ резервации, номер сервера (хоста) через который происходило взаимодействие и ID пути. Эти данные полностью характеризуют клиента. Ключ резервации является ключом, а остальное является значением для синхронизирующего модуля.

Другие команды SCSI, которые работают с некоторым состоянием,

могут быть реализованы в кластерном режиме аналогично. Всё, что для этого требуется — это представить состояние, как набор сущностей, связанных, как ключ-значение, после чего синхронизировать состояние с помощью синхронизирующего модуля.

5.4. Команда «Compare and Write»

Команды SCSI, которые не работают с внутренним состоянием, реализуются в кластерном режиме еще проще: единственное в чем нужно убедиться это то, что одновременное выполнение команд на разных узлах кластера не приведет к неконструктивному состоянию. Этого можно добиться, если сделать некоторые команды атомарными[5] с точки зрения кластера. Атомарность с точки зрения кластера, означает, что у команды могут быть этапы выполнения на конкретном узле, но команда не может быть частична применена на всем кластере.

Чтобы добиться атомарности команды можно использовать блокировки, которые нам предоставляет синхронизирующий модуль. Должны быть выделены ресурсы, с которыми работает команда, и зарегистрированы в синхронизирующем модуле некоторые связанные с этими ресурсами сущности. При поступлении команды сначала берутся блокировки на все необходимые ресурсы, потом команда выполняется, а потом блокировки снимаются. Это напоминает использование семафоров, но только на уровне кластера.

Проблем с взаимоблокировками не будет, так как запросы на взятие блокировок попадают в очереди DLM, где работает механизм определения и предотвращения взаимоблокировок. Нам остается только при обработке результата запроса на блокировку, обработать код ошибки связанный с взаимоблокировкой. А поскольку в стандарте не описано, какая команда должна выполниться первой при одновременном поступлении нескольких команд в систему, то нам не нужно заботиться о проблеме «гонок» в данном случае. Важно лишь то, что команды применяются последовательно и после отправки статуса команды на SCSI-

[5]Напомним, что атомарность в данной работе используется, как термин из описания работы баз данных, а не процессора.

initiator результат команды должен быть глобально консистентен, как для инициаторов так и для узлов кластера.

В рамках данный работы был реализован кластерный режим одной из атомарных команд протокола SCSI. Целью данный работы не является реализация всех команд протокола SCSI в кластерном режиме, так как SCSI включает в себя сотни команд, а его исходный код занимает сотни тысяч строк кода, поэтому реализация кластерного режима во всех командах SCSI это отдельная серьезная задача.

Тем не менее был реализован кластерный режим в одной из команд, что является подтверждением правильности концепта. Выбор пал на команду Compare And Write, поскольку с помощью этой команды можно синхронизировать любые другие команды.

Команда Compare and Write (CAW) протокола SCSI является аналогом распространенного механизма Compare and Swap, который реализован во многих языках программирования, например в C++[24]. Команда CAW заключается в том, что происходит сравнение значения по некоторому адресу со значением аргументом, если они не равны, то команда завершается. Если же значения равны, то производится запись значения другого аргумента по указанному адресу. Данная команда может использоваться, как кластерный мьютекс, например если считать значение 0, как показатель свободности некоторого ресурса. Один узел может проверить значение на равенство нулю, и в случае равенства записать 1, тем самым пометить, что ресурс занят. Если данная команда не будет атомарной, то сразу несколько узлов могут прочитать 0, прежде чем один из них запишет 1. Это приведет к тому, что более одного узла начнут пользоваться неразделяемым ресурсом, что приведет к ошибке. Чтобы избежать этого команда CAW должна быть атомарной.

Атомарность команды была реализована согласно механизму, описанному ранее, но более подробно это выглядит так:

1) Вызывается функция синхронизирующего модуля: create_key(). В качестве ключа передается значение адреса, по которому происходит

чтение. Если такой ключ уже есть, ничего не произойдет, если его нет, то ключ создается.

2) Вызывается функция get_lock(). В качестве ключа передается тот же параметр, как в пункте 1. Данная функция будет ожидать, если другой узел уже взял блокировки на данный ресурс. Если ресурс свободен, функция берет блокировку на ресурс.

3) Когда блокировка взята, происходит чтение, сравнение и если необходимо запись.

4) Вызывается функция release_lock(), освобождающая ресурс.

Так как функция get_lock() атомарна, то в критической секции (пункт 3) могут находиться не больше одного узла. Таким образом и команда Compare And Write становится атомарной.

6. Тестирование

Можно выделить три части в написанном коде: синхронизирующий модуль, драйвер символьного устройства и изменения, сделанные в SCSI-target (команда Compare And Write (CAW)).

Для проведения тестирования команды CAW модули SCSI-target собирались и загружались в ядро Linux версии 5.8.0-50-generic. Также загружался синхронизирующий модуль. Это выполнялось на всех узлах кластера. После чего с помощью Pacemaker'а стартовался кластер (использовалась консольная утилита pcs). На одном из узлов настраивался диск, доступный по протоколу iSCSI (с помощью утилиты targetcli). Затем с помощью функции sg_compare_and_write пакета sg3_utils на всех узлах вызывалась команда CAW , примененная к одному и тому же блоку диска. После этого проверялось, что в критической секции команды CAW всегда находилось не больше одного узла[6].

Драйвер символьного устройства тестировался вместе с синхронизирующим модулем, поскольку сложно тестировать драйвер обособленно, так как операции записи должны вызывать команды синхронизирующего модуля, а операции чтения отображают состояние, которое будет оставаться пустым без синхронизирующего модуля. Вызов функций синхронизирующего модуля из драйвера прозрачен и для контроля правильности этого этапа все шаги логируются в лог ядра.

Тестирование синхронизирующего модуля выполнялось следующим образом: запускались несколько виртуальных машин с помощью программы VirtualBox. Узлы объединялись в кластер с помощью pacemaker'а. Затем на одном из узлов запускались тесты. Тесты написаны с использованием фреймворка PyTest[25], причем благодаря secure shell (ssh) соединению, команды могут вызываться не только на том узле на котором запущен тест, но и на других узлах кластера. Тест проверят состояния на всех узлах кластера, также с использованием ssh. Для запуска команд и проверки состояния использовался разработанной в рамках этой

[6]Для этого в начале критической секции стоял таймаут на несколько секунд, а входы и выходы из критической секции логировались.

работы драйвер символьного устройства.

Также в рамках данной работы не производилась оценка производительности модифицированных команд и синхронизирующего модуля, так как не стояло задачи предоставить оптимальное по времени выполнения решение.

7. Заключение

Полученные результаты:

В ходе выполнения данной работы были достигнуты следующие результаты:

- проведено исследование предметной области, изучены существующие стандарты;
- произведен обзор решений, которые позволяют организовать SCSI кластер и зафиксированы лучшие аспекты этих решений;
- разработан метод реализации кластерного механизма в SCSI-target ядра Linux;
- реализован разработанный метод[7], с помощью которого добавлен кластерный режим в команду SCSI Compare And Write[8];
- проведено тестирование ключевой функциональности с помощью тестовых стендов.

Благодарность: Я выражаю большую благодарность моему консультанту из компании Yadro — Шелехину К. А., который помогал мне на протяжении всей работы над дипломом как технически: пояснял непонятные для меня аспекты работы с ядром Linux и кластерным ПО и проводил ревью кода, так и морально: выражал поддержку и давал общие советы.

[7] https://github.com/Dmitree-Max/DLM_SCSI

[8] https://github.com/Dmitree-Max/SCSI_target_clustered

Список литературы

[1] Co. Hewlett-Packard. Small Computer Systems Interface (SCSI) // documentation. — URL: https://tools.ietf.org/html/rfc3783 (дата обращения: 8.10.2020).

[2] Kernel.org. TCM Userspace Design // Статья. — URL: https://www.kernel.org/doc/html/latest/target/tcmu-design.html (дата обращения: 09.10.2020).

[3] Felixcloutier. CMPXCHG — Compare and Exchange // documentation. — URL: https://www.felixcloutier.com/x86/cmpxchg (дата обращения: 22.03.2020).

[4] Linux-iscsi.org. Targetcli // Официальный сайт. — URL: http://linux-iscsi.org/wiki/Targetcli (дата обращения: 11.11.2020).

[5] Duncan Lee. open-iscsi // Github. — URL: https://github.com/open-iscsi/open-iscsi (дата обращения: 11.11.2020).

[6] Virtualbox.org. Welcome to VirtualBox.org // Официальный сайт. — URL: https://www.virtualbox.org (дата обращения: 02.02.2021).

[7] T10. SCSI Primary Commands - 4 // RFC. — 2020. — URL: https://www.t10.org/members/w_spc4.html (дата обращения: 5.11.2020).

[8] T10/BSR. Information technology - SCSI Block Commands – 3 (SBC-3).

[9] T10/BSR. Information technology - SCSI Architecture Model - 5 (SAM-5).

[10] Sourceforge. Generic SCSI Target Subsystem for Linux // paper. — 2020. — URL: http://scst.sourceforge.net (дата обращения: 24.10.2020).

[11] lwn. A tale of two SCSI targets // Article. — 2019. — URL: https://lwn.net/Articles/424004/ (дата обращения: 12.11.2020).

[12] Sourceforge. Features comparison between Linux SCSI targets // paper. — 2020. — URL: http://scst.sourceforge.net/comparison.html (дата обращения: 24.10.2020).

[13] Oracle. Distributed Lock Manager: Access to Resources // documentation. — 2019. — URL: https://docs.oracle.com/cd/A57673_01/DOC/server/doc/SPS73/chap8.htm (дата обращения: 12.10.2020).

[14] FreeBSD. FreeBSD CTL // documentation. — URL: https://www.freebsd.org/cgi/man.cgi?query=ctl&sektion=4 (дата обращения: 8.11.2020).

[15] Oracle. OCFS2 Best Practices Guide // documentation. — URL: https://www.oracle.com/us/technologies/linux/ocfs2-best-practices-2133130.pdf (дата обращения: 9.11.2020).

[16] Clusterlabs. Pacemaker Cluster Resource Manager // Официальный сайт. — URL: https://www.clusterlabs.org/pacemaker/ (дата обращения: 12.12.2020).

[17] Hellwig Christoph. Add support for reservations to NVMeoF target // Patch request. — 2017. — URL: http://lists.infradead.org/pipermail/linux-nvme/2017-August/012592.html (дата обращения: 5.4.2021).

[18] Thomas Kristin. Programming Locking Applications // book. — 2001. — URL: http://opendlm.sourceforge.net/cvsmirror/opendlm/docs/dlmbook_final.pdf (дата обращения: 22.12.2020).

[19] Linux. dlm_lock(3) - Linux man page // documentation. — URL: https://linux.die.net/man/3/dlm_lock (дата обращения: 02.03.2020).

[20] Oracle. Distributed Lock Manager: Access to Resources // documentation. — URL: https://docs.oracle.com/cd/A57673_01/DOC/server/doc/SPS73/chap8.htm (дата обращения: 7.12.2020).

[21] Kernel.org. Completions - “wait for completion” barrier APIs // documentation. — URL: https://www.kernel.org/doc/html/latest/scheduler/completion.html (дата обращения: 28.01.2020).

[22] Kernel.org. Concurrency Managed Workqueue // documentation. — 2010. — URL: https://www.kernel.org/doc/html/latest/core-api/workqueue.html (дата обращения: 3.03.2021).

[23] lwn. Char Drivers // book. — 2005. — URL: https://static.lwn.net/images/pdf/LDD3/ch03.pdf (дата обращения: 17.12.2020).

[24] Cppreference. C++ Atomic operations library // documentation. — 2019. — URL: https://en.cppreference.com/w/cpp/atomic/atomic_compare_exchange (дата обращения: 12.04.2020).

[25] Pytest.org. Full pytest documentation // documentation. — URL: https://docs.pytest.org/en/stable/contents.html (дата обращения: 21.04.2020).