Министерство науки и высшего образования Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

ИССЛЕДОВАНИЕ БАЗОВЫХ АЛГОРИТМОВ И СТРУКТУР ДАННЫХ ДЛЯ NON-VOLATILE MEMORY

Автор: Кокорин Илья Всеволодович _____

Направление подготовки: 01.03.02 Прикладная математика и информатика

Квалификация: Бакалавр

Руководитель ВКР: Аксёнов В.Е., к.т.н. _____

Санкт-Петербург, 2020 г.

Обучающийся Кокорин Илья Всеволодович
Группа М3439 Факультет ИТиП

Направленность (профиль), специализация
Математические модели и алгоритмы в разработке программного обеспечения

ВКР принята «____» _____ 20___ г.

Оригинальность ВКР _____%

ВКР выполнена с оценкой _____

Дата защиты «____» _____ 20___ г.

Секретарь ГЭК Павлова О.Н. _____

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

проф., д.т.н. Парфенов В.Г. _____

«____» _____ 20___ г.

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ

Обучающийся Кокорин Илья Всеволодович

Группа М3439 **Факультет** ИТиП

Квалификация: Бакалавр

Направление подготовки: 01.03.02 Прикладная математика и информатика

Направленность (профиль) образовательной программы: Математические модели и алгоритмы в разработке программного обеспечения

Тема ВКР: Исследование базовых алгоритмов и структур данных для non-volatile memory

Руководитель Аксёнов В.Е., к.т.н., научный сотрудник ФИТиП

2 Срок сдачи студентом законченной работы до: «____» _____ 20___ г.

3 Техническое задание и исходные данные к работе

Требуется разработать алгоритм запуска программ для энергонезависимой байтоадресуемой памяти с произвольным доступом в модели Nesting-safe recoverable linearizability. Полученный алгоритм должен исполнять программы для энергонезависимой байтоадресуемой памяти и восстанавливать систему после сбоя. Кроме того, должна быть возможность эмулировать данный алгоритм и запускать программы для энергонезависимой байтоадресуемой памяти с произвольным доступом на системах, в которых в качестве энергонезависимого хранилища используются жёсткие диски.

4 Содержание выпускной квалификационной работы (перечень подлежащих разработке вопросов)

Необходимо изучить архитектуру энергонезависимой байтоадресуемой памяти с произвольным доступом и примеры алгоритмов для этой памяти, а также разработать алгоритм исполнения программ в этой памяти. После этого необходимо эмулировать разработанный алгоритм с помощью системы, использующей жёсткие диски в качестве энергонезависимого хранилища и протестировать с его помощью алгоритм CAS.

5 Перечень графического материала (с указанием обязательного материала)

Графические материалы и чертежи работой не предусмотрены

6 Исходные материалы и пособия

а) Статья [2]

б) Документация библиотеки `PMDK`

7 Дата выдачи задания «01» сентября 2019 г.

Руководитель ВКР _____

Задание принял к исполнению _____ «01» сентября 2019 г.

Министерство науки и высшего образования Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

АННОТАЦИЯ
ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ

Обучающийся: Кокорин Илья Всеволодович

Наименование темы ВКР: Исследование базовых алгоритмов и структур данных для non-volatile memory

Наименование организации, в которой выполнена ВКР: Университет ИТМО

ХАРАКТЕРИСТИКА ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ

1 Цель исследования: Разработать алгоритм исполнения программ в энергонезависимой памяти с произвольным доступом (NVRAM)

2 Задачи, решаемые в ВКР:

а) Изучить архитекуру энергонезависимой памяти с произвольным доступом (NVRAM)
б) Познакомиться с примерами конкурентных алгоритмов для NVRAM
в) Разработать алгоритм исполнения конкуренных программ для NVRAM и восстановления системы после сбоя
г) Эмулировать разработанный алгоритм с помощью системы, использующей жёсткие диски в качестве энергонезависимого хранилища
д) Протестировать с его помощью изученный ранее алгоритм CAS для NVRAM

3 Число источников, использованных при составлении обзора: 7

4 Полное число источников, использованных в работе: 29

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
0	0	0	24	1	4

6 Использование информационных ресурсов Internet: да, число ресурсов: 19

7 Использование современных пакетов компьютерных программ и технологий:

Пакеты компьютерных программ и технологий	**Раздел работы**
Библиотека `PMDK` для работы с энергонезависимой памятью	Реализация системы
Интегрированная среда разработки `CLion`	Реализация системы
Распределённая система контроля версия `git`	Реализация системы
Компилятор `GNU C++` языка `C++`	Реализация системы

8 Краткая характеристика полученных результатов

Удалось разработать алгоритм запуска программ для NVRAM, который можно использовать как для запуска программ в системе с использованием NVRAM, так и в системе с использованием жёстких дисков в качестве энергонезависимого хранилища. Разработанный алгоритм был эмулирован на жёстких дисках с использованием библиотеки PMDK.

9 Гранты, полученные при выполнении работы

Грантов при выполнении работы получено не было.

10 Наличие публикаций и выступлений на конференциях по теме выпускной работы

Победа в конкурсе докладов IX Конгресса Молодых Учёных

Обучающийся Кокорин И.В. _____

Руководитель ВКР Аксёнов В.Е. _____

«____» _____ 20___ г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Целью работы была разработка алгоритма исполнения программ в энергонезависимой памяти с произвольным доступом (далее NVRAM).

Передо мной стояли следующие задачи:

— Изучить архитекуру энергонезависимой памяти с произвольным доступом (NVRAM).
— Познакомиться с примерами конкурентных алгоритмов для NVRAM.
— Разработать алгоритм исполнения конкурентных программ для NVRAM и для восстановления системы после сбоя.
— Проэмулировать разработанный алгоритм с помощью системы, использующей жёсткие диски в качестве энергонезависимого хранилища.
— Протестировать с его помощью изученный ранее алгоритм CAS для NVRAM.

Актуальность задачи обусловлена существованием нескольких конкурентных алгоритмов для NVRAM, но отсутствием алгоритма их запуска и восстановления системы после сбоя. Это делает невозможным запуск и тестирование алгоритмов для NVRAM на реальных системах.

Новизна работы обусловлена тем, что современные теоретические модели не учитывают архитектуру реальных систем. Например, подразумеватся отсутствие энергозависимых кэшей и регистров и наличие энергнезависимого стека, который мог бы сохранять состояние системы даже если в системе произошёл сбой. Например, в работе [2] подразумевается отсутствие кэшей и наличие энергонезависимого стека.

Поэтому новизна работы состоит в разработке алгоритма запуска программ для NVRAM на реальном железе с учётом всех его особенностей.

Практическая значимость работы обусловлена тем, что представленное сегодня на рынке железо с NVRAM очень дорогое. Несмотря на это, существует потребность в тестировании алгоритмов для NVRAM. Если удастся эмулировать алгоритм запуска программ для NVRAM на более дешёвом железе (например, на жёстких дисках), это позволит тестировать алгоритмы для NVRAM даже без доступа к дорогому железу.

Работа имеет следующую структуру:

— В первой главе работы проводится анализ предметной области: рассматривается архитектура NVRAM, модели исполнения программ в системе с разделяемой энергонезависимой памятью, модели сбоев в такой системе и условия корректности исполнений в такой модели.
— Во второй главе разрабатывается алгоритм запуска программ на системе с NVRAM и восстановления системы после сбоя. Кроме того, рассматриваются вопросы: 1) корректной работы с адресами функций и указателями на NVRAM, 2) возврата значений из функций, запуска разработанного алгоритма на системе, использующей жёсткие диски в качестве энергонезависимого хранилища, и 3) использования разработанного алгоритма для запуска программ, удовлетворяющих различным условиям корректности, например, линеаризуемости.
— В третьей главе рассматривается задача выделения памяти в куче NVRAM и представляются два достаточно простых алгоритма выделения памяти. Рассматриваются плюсы и минусы каждого из них, производится выбор подходящего нам алгоритма выделения памяти в куче NVRAM.
— В четвёртой главе рассматривается применение разработанного в предыдущих главах решения для запуска и тестирования на корректность алгоритма CAS для NVRAM, описанного в статье [2].

ГЛАВА 1. ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ

1.1. Архитектура NVRAM

Архитектура NVRAM имеет следующие отличительные особенности:

— Энергонезависимость, как у жёстких дисков. Это значит, что записанная в ней информация остаётся доступной даже в случае сбоя системы, и эту информацию можно использовать для восстановления системы после сбоя.

— Быстрый произвольный доступ с побайтовой адресацией, подобно оперативной памяти. Это позволяет придумывать для систем с NVRAM потенциально более эффективные алгоритмы восстановления системы после сбоя, чем традиционные алгоритмы на основе append-only журналирования, используемые на системах с жёсткими дисками. Для систем с жёсткими дисками такие алгоритмы, вероятно, будут неприменимы из-за низкой скорости случайного доступа, прямо вытекающей из механической природы жёстких дисков, но системы с NVRAM обеспечивают гораздо более быстрый случайный доступ и, следовательно, не имеют таких ограничений.

— Наличие в системе с NVRAM энергозависимой памяти. Например, кэши NVRAM и регистры всегда остаются энергозависимыми, вследствие чего при сбое системы теряются результаты последних вычислений (которые на x86 велись в регистрах) и данные, которые должны были быть записаны в NVRAM, но не успели записаться и попали в кэш. Для борьбы с потерей кэшированных данных системы с NVRAM предоставляют возможность явно потребовать сброса кэшей NVRAM в энергонезависимую память.

— Сброс данных в NVRAM производится кэш-линиями, при этом запись одной кэш-линии атомарна, то есть в случае сбоя в процессе записи вся кэш-линия либо целиком записывается, либо целиком не записывается в NVRAM. Запись, например, половины кэш-линии невозможна. Однако запись большого количества данных, не влезающих в одну кэш линию, происходит неатомарно, так как сбой может произойти после записи первой кэш-линии, но до записи всех остальных. В таком случае первая кэш-линия будет записана в NVRAM, а остальные данные не будут записаны.

1.2. Модель исполнения

В этой секции будет описана модель исполнения в системе с общей памятью, которую я буду использовать в работе. Эта модель является дополненной версией модели, описанной в статье [2].

В системе существует n процессов $\{p_i\}_{i=1}^{n}$, исполняющихся одновременно и асинхронно, и m расположенных в общей энергонезависимой памяти объектов $\{O_j\}_{j=1}^{m}$. Процессы взаимодействуют друг с другом, вызывая различные операции на объектах в общей памяти (см. Рисунок 1).

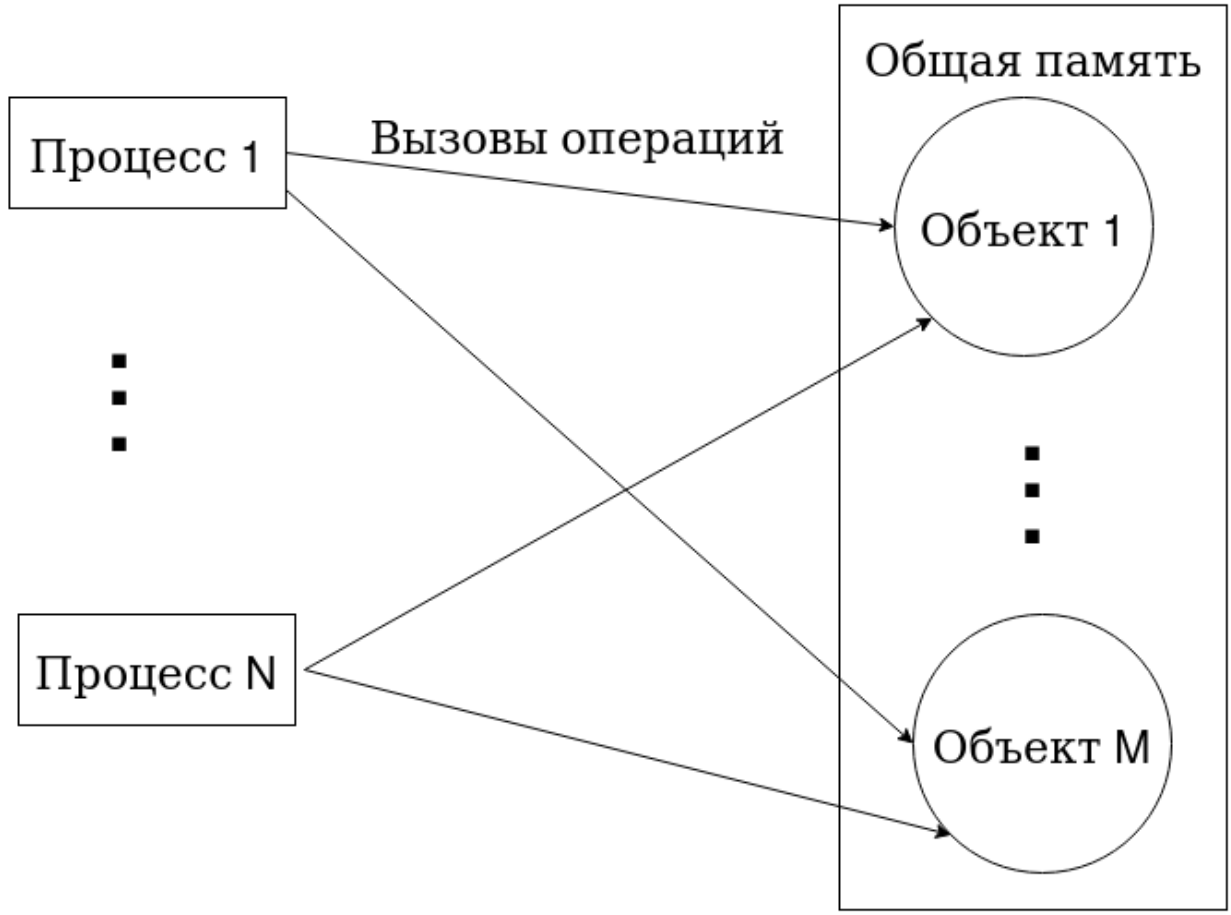

Рисунок 1 – Процессы и общие объекты

Расположенные в общей памяти объекты могут поддержживать операции чтения (R), записи (W) и чтения-модификации-записи (RMW).

Операцией чтения-модификации-записи (RMW) называется операция, которая атомарно исполняет следующие операции:

— Читает записанное в ячейке памяти значение.

— Изменяет его неким образом (например, увеличивая на один).

— Записывает значение, полученное после изменения, обратно в ячейку памяти.

Примерами таких операций являются `compare-and-swap` [3], `fetch-and-add` [5], `test-and-set` [25].

В работе [8] описано определение RMW-операции, примеры различных RMW-операций и их классификация.

В рассматриваемой модели вся разделяемая память является энергонезависимой, но подразумевается существование в системе также и энергозависимой памяти. Каждая ячейка энергозависимой памяти reg_j является локальной для какого-то процесса p_i, это означает, что доступ к reg_j может получать только процесс p_i и ни один процесс, кроме p_i.

Таким образом, помимо доступа к разделяемым объектам, каждый процесс имеет доступ к локальным объектам, доступным только ему и недоступным другим процессам. Эти объекты хранятся в энергозависимой памяти данного процесса и поддерживают операции чтения (R) и записи (W) (см. Рисунок 2).

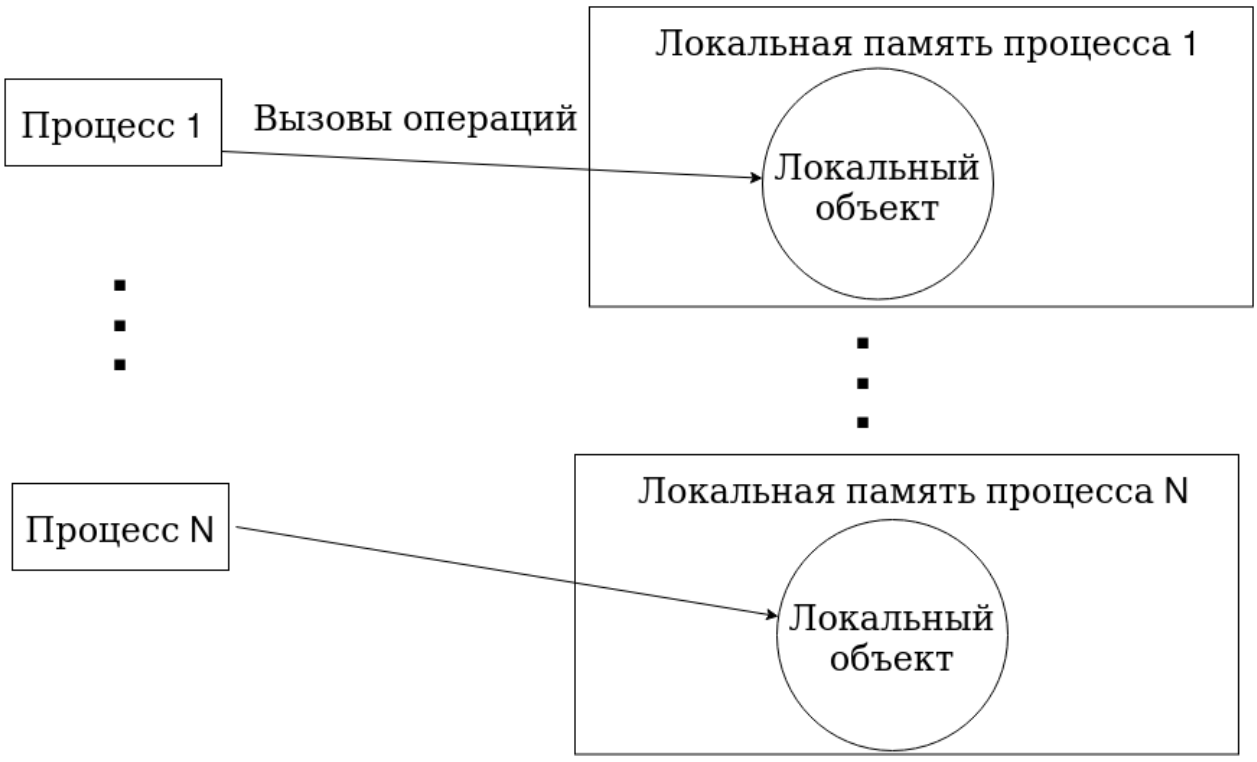

Рисунок 2 – Процессы и локальные объекты

При этом нам не важно, где именно расположены локальные объекты: в регистрах процессора, на стеке или в динамической памяти. Нас интересует только то, что все локальные объекты:

— Расположены в энергозависимой памяти (а значит, их значение теряется при сбое системы - более подробно это свойство локальный объектов будет обсуждаться в секции 1.3).
— Доступны только одному процессу.

Рассмотренная модель не учитывает некоторые особенности реального железа: например, в ней не учитывается наличие энергозависимых кэшей NVRAM и возможность наличия в системе разделяемой энергозависимой памяти.

1.3. Модели сбоев

Существуют две модели сбоев в системе.

— Individual crash-failure.

  Эта модель описана в работе [2]. В такой модели каждый процесс может может столкнуться со сбоем в процессе работы независимо от остальных процессов системы. При сбое процесс перестаёт работать до тех пор, пока он не будет перезапущен. Перезапуск процесса восстанавливает его работоспособность, но при этом теряется значение всех локальных переменных, расположенных в энергозависимой памяти перезапускаемого процесса.

  Записанные же в NVRAM данные общих объектов не теряются и остаются доступными процессу после восстановления работы.

  Так как сбойный процесс неотличим от очень медленного, система при сбоях сохраняет гарантии прогресса, которым она удовлетворяла до сбоя. Например, lock-free алгоритм остаётся таковым, даже если один или несколько процессов системы столкнулись со сбоем в ходе работы.

— System crash-failure.

  В этой модели сбой происходит не в одном процессе, а во всей системе сразу. Система перестаёт работать до тех пор, пока она не будет перезапущена целиком. При этом после восстановления каждый процесс системы теряет данные, записанные в его локальной энергозависимой памяти. Следовательно, в такой модели при сбое системы теряются все данные, записанные в энергозависимой памяти.

Как и в предыдущей модели, после сбоя системы данные общих объектов, записанные в NVRAM, не исчезают при сбое, и после возобновления работы эти данные продолжают быть доступны всем процессам системы. После возобновления работы эти данные могут быть использованы для восстановления после сбоя.

Заметим, что модель System crash-failure является частным случаем модели Individual crash-failure, так как сбой всей системы может быть представлен как одновременный сбой каждого из n индивидуальных процессов $\{p_i\}_{i=1}^{n}$.

Несмотря на то, что модель Individual crash-failure является более общей моделью, в работе будем рассматривать модель System crash-failure, так как она более точно описывает процесс и последствия сбоя реальной системы. В реальной системе сложно себе представить ситуацию, при которой сбой происходит в одном процессе, но легко представить себе ситуацию, при которой сбой происходит во всех процессах сразу — это происходит, например, при выключении питания. Кроме того, на реальных системах проэмулировать сбой в модели System crash-failure гораздо проще.

1.4. Исполнение операций

Модель конкуррентного исполнения операций в NVRAM описана в статье [2].

Каждой операции `Op`, которая может быть исполнена в системе, сопоставлена операция `Op.Recover`. `Op.Recover` получает при вызове те же аргументы, что и `Op` и исполняется на том же объекте, что и `Op`. `Op.Recover` должна завершить операцию `Op` в случае, если в процессе исполнения `Op` произойдёт сбой системы.

Исполнением системы H называется последовательность шагов $\{step\}_{i=1}^{Z}$, при этом каждый шаг $step_i$ имеет один из четырёх следующих типов:

- `INV(p, O, Op, args)` — вызов процессом `p` операции `Op` на объекте `O` с аргументами `args`
- `RES(p, O, Op, ret)` — завершение операции `Op`, вызванной на объекте `O` процессом `p`, результатом операции является `ret`.
- `CRASH` — сбой системы
- `RECOVER` — восстановление системы после сбоя.

Операция `Op` называется исполняющейся на процессе `p`, если в этом процессе уже произошёл вызов этой операции, но ещё не произошёл возврат из неё. У процесса `p` в каждый момент может быть несколько исполняющихся операций одновременно (это происходит, например, если в ходе исполнения операции `F` происходит вызов операции `G`). В таком случае в каждом процессе исполняющиеся операции формируют вложенную последовательность (например, для примера выше будем говорить, что исполнение операции `G` вложено в исполнение операции `F`) (см. Рисунок 3).

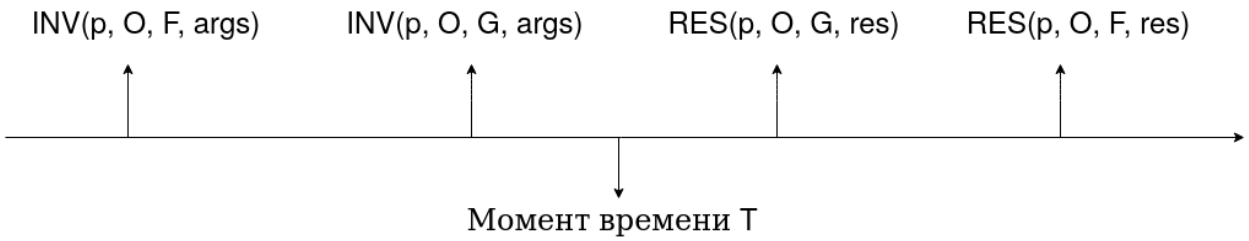

Рисунок 3 – В момент времени T на процессе p одновременно исполняются операции F и G, причём исполнение G вложено в исполнение F

Для восстановления системы после сбоя нужно для каждого процесса системы `p` вызвать `recover`-версию каждой операции, исполнявшейся в этом процессе в момент сбоя, начиная с той операции, которая была вызвана последней и заканчивая той, которая была вызвана первой. Например, в примере выше сначала будет вызвана операция `G.Recover`, а после её завершения `F.Recover` (см. Рисунок 4).

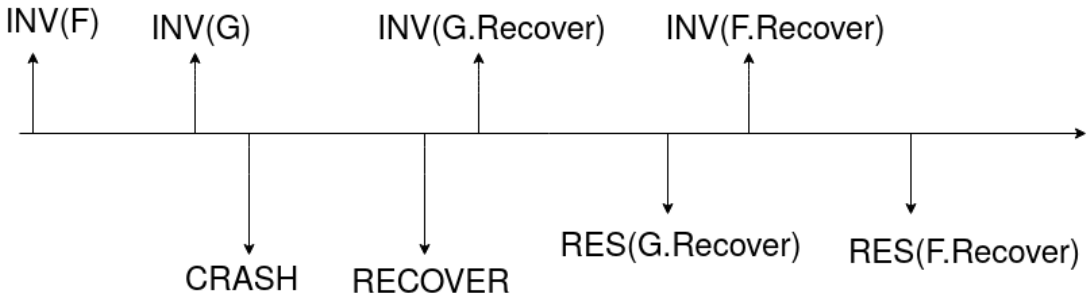

Рисунок 4 – Восстановление системы после сбоя

1.5. Условия корректности

Существует несколько условий корректности конкурентных алгоритмов для NVRAM. Перечислим их в порядке убывания строгости.

— Nesting-safe recoverable linearizability описано в статье [2]. Это условие корректности требует, чтобы любая операция в системе была завершена даже в том случае, если в системе произошёл сбой. Для завершения операций, которые исполнялись в момент сбоя необходимо после восстановления запускать recover-версии операций, исполнявшихся в момент сбоя.
— Durable linearizability описано в статье [10]. Это условие корректности требует, чтобы эффект всех операции, исполнение которых завершилось до сбоя, был виден после восстановления. Если же операция исполялась в момент сбоя, то её эффект может быть как виден после восстановления, так и не виден.
— Buffered durable linearizability является ослабленной версией durable linearizability и описано в статье [1]. Это условие корректности отличается от последнего тем, что не требует после восстановления видимости эффектов всех операций, исполнение которых завершилось до сбоя. Это условие корректности допускает, что эффект только части таких операций будет виден после восстановления. Однако, это условие корректности требует, чтобы каждый объект предоставлял операцию `sync`. Если на объекте была вызвана операция `sync` и её исполнение завершилось до сбоя, то после восстановления должны быть видимы результаты всех операций, исполнение которых завершилось до вызова операции `sync`.

В этой работе мы предоставляем алгоритм запуска программ, удовлетворяющих условиям Nesting-safe recoverable linearizability.

1.6. Описание существующих решений поставленной проблемы

На текущий момент не существует алгоритмов запуска программ на системах с NVRAM, которые можно было бы эмулировать на системах, использующих жёсткие диски в качестве энергонезависимого хранилища, и использовать для тестирования алгоритмов для NVRAM даже без доступа к железу с NVRAM.

Выводы по главе 1

В этой главе был проведён анализ предметной области: рассмотрена архитектура NVRAM, модели исполнения в системе с разделяемой энергонезависимой памятью, модели сбоев в такой системе, а также условия коррект-

ности таких исполнений. Кроме того, было определено понятие `recover`-операции, которое играет важную роль в следующих главах. Был проведён анализ существующих решений и показано, что разработанное решение является единственным в своём роде.

ГЛАВА 2. РАЗРАБОТКА СИСТЕМЫ ИСПОЛНЕНИЯ И ВОССТАНОВЛЕНИЯ

2.1. Процессы и потоки

В теории конкуррентного программирования зачастую термины ‘‘процесс’’ и ‘‘поток’’ не различают и считают синонимами. Однако, при реализации алгоритмов очень важно понимать разницу между процессами и потоками, поэтому в дальнейшем будем их различать.

Будем использовать терминологию операционной системы Linux и считать, что каждый процесс [23]

— исполняется в собственном адресном пространстве;
— имеет иллюзию монопольного использования всей памяти системы (такая иллюзия достигается при помощи использования механизма виртуальной памяти, как это, например, описано в статье [4]);
— состоит из нескольких потоков.

Все потоки [26] одного процесса разделяют единое адресное пространство этого процесса и, следовательно, имеют общую кучу и общую статическую память. Например, из этого следует, что все потоки одного процесса разделяют единый сегмент кода. Но каждый поток системы:

— имеет собственное процессорное время;
— имеет собственный стек;
— имеет иллюзию монопольного использования регистров процессора — например, наличие у каждого потока независимого от остальных потоков регистра `RIP` позволяет всем потокам независимо исполнять код из сегмента кода процесса, поскольку каждый поток имеет собственный указатель на текущую исполняемую в потоке инструкцию [24].

Несмотря на то, что в большинстве случаев мы считаем, что все потоки одного процесса разделяют общую память, а различные процессы не разделяют память вообще, из этого правила бывают исключения. Потоки могут использовать thread-local storage [27] для хранения статических данных, не разделяемых между потоками одного процесса, а доступных только одному потоку. Процессы же могут разделять память с другими процессами с использованием различных механизмов межпроцессного взаимодействия, например shared memory mapped files [15].

Использование процессов и потоков в операционной системе Linux описано, например, в книге [11].

Заметим, что сущности, называемые на практике (и, следовательно, в дальнейшем тексте) ‘‘потоками’’ соответствуют сущностям, называемым в теории конкуррентного программирования (и, следовательно, в первой главе) ‘‘процессами’’.

2.2. Взаимодействие с NVRAM

Считаем, что NVRAM предоставляет нам следующие функции:

— `void pmem_read(const void* src, void* dest, size_t len)` копирует `len` байт из `src` (этот указатель должен указывать на участок NVRAM) в `dest` (этот указатель должен указывать на участок энергозависимой памяти) и, тем самым, позволяет читать данные из NVRAM в энергозависимую память.

— `void pmem_write(void* dest, const void* src, size_t len)` копирует `len` байт из `src` (этот указатель должен указывать на участок энергозависимой памятм) в `dest` (этот указатель должен указывать на участок NVRAM) и, тем самым, позволяет писать данные из энергозависимой памяти в NVRAM. Заметим, что данные после записи могут не попасть в NVRAM непосредственно, а остаться в кэше.

— `void pmem_flush(const void* dest, size_t len)` производит сброс энергозависимых кэшей в NVRAM, начиная с адреса `dest`. При этом сброшено будет не менее `len` байт. Следовательно, после завершения этого вызова все изменения, сделанные на интервале $[dest; dest + len)$ будут сброшены в NVRAM из кэша. При этом В NVRAM может быть сброшено более `len` байт, так как сброс данных в NVRAM производится кэш-линиями. Например, невозможно произвести сброс одного байта в NVRAM, необходимо сбросить всю кэш-линию, в которой находится этот байт. После завершения вызова в NVRAM из кэша будут сброшен интервал $[left; right)$, при этом $[dest; dest + len) \subseteq [left; right)$, где $left$ - адрес начала кэш-линии.

2.3. Персистентный стек

Заметим, что для запуска программ на NVRAM, необходимо поддерживать информацию о том, какие операции исполнялись в момент сбоя в каждом

потоке. Кроме того, необходимо поддерживать информацию о порядке их вызова для того, чтобы знать, в каком порядке запускать recover-операции.

Для того, чтобы поддерживать информацию о порядке вызова исполнявшихся в момент сбоя функций, необходимо просто для каждого потока поддерживать его стек, состоящий из стековых фреймов, по одному на каждую вызванную функцию. При вызове функции мы добавляем новый фрейм на вершину стека, при возврате из функции стираем последний стековый фрейм с вершины стека. Таким образом, в момент сбоя на стеке потока t_i оказываются расположены стековые фреймы тех и только тех функций, которые исполнялись на этом потоке в момент сбоя. Причём, чем ближе стековый фрейм, соответствующий некой функции, расположен к вершине стека, тем позже в потоке была вызвана эта функция (см. Рисунок 5).

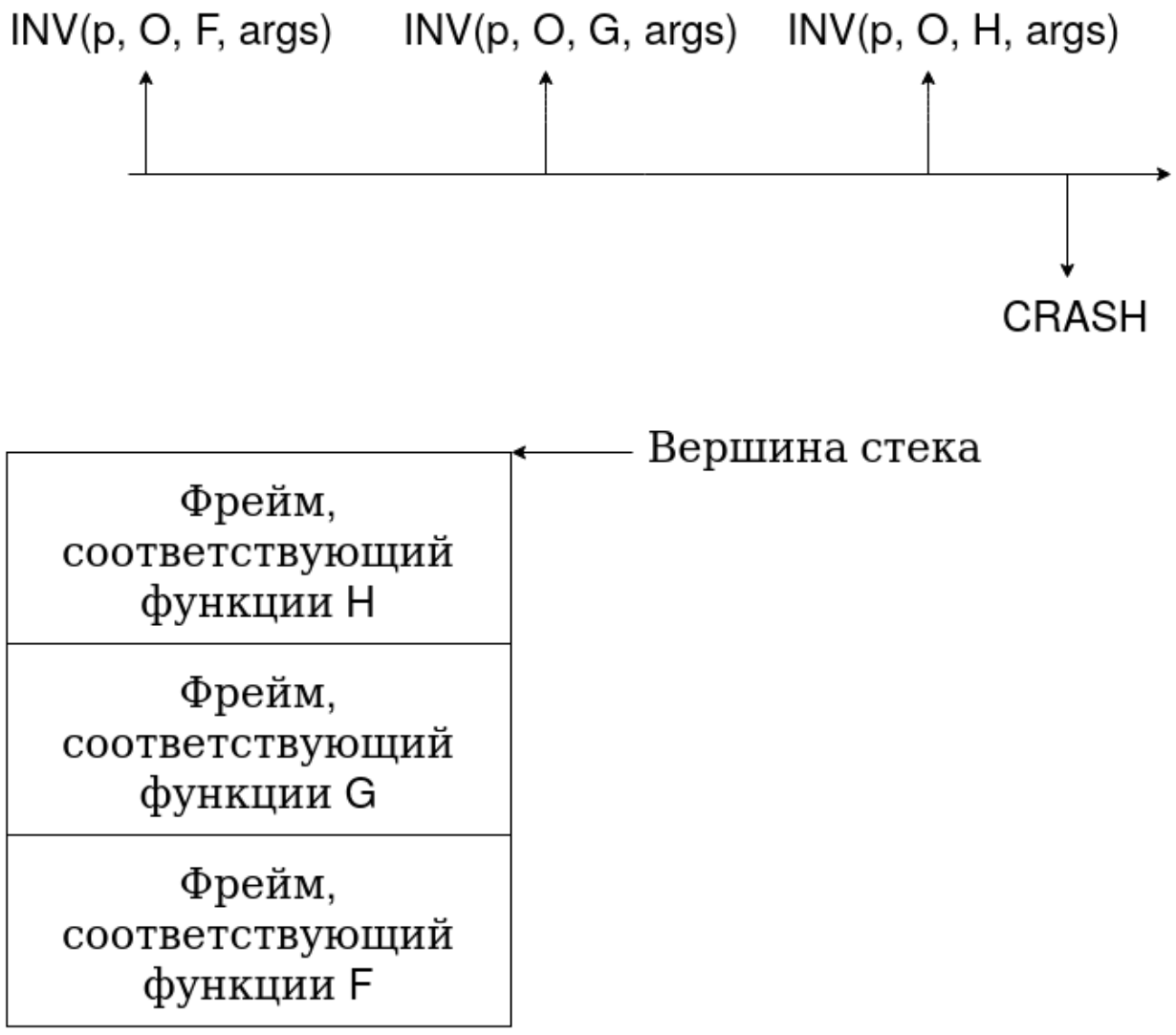

Рисунок 5 – Исполнение и соответствующий этому исполнению стек потока p

Для каждого потока операционная система итак поддерживает собственный стек вызовов. Проблема состоит в том, что стек вызовов выделяется в энергозависимой области памяти, а простой его перенос в энергонезависимую память проблему не решает.

Рассмотрим, например, передачу адресов возврата через стек вызовов. Если функция `f()` хочет вызвать функцию `g()`, вызывающая функция пишет на стек содержимое регистра `RIP` (указывающего на адрес команды, следующей за исполняющейся в текущий момент, то есть на адрес команды, следующей за командой `CALL g`) и передаёт управление вызываемой функции. Вызываемая функция, завершив работу, снимает со стека записанное там значение `RIP` и делает `JMP` на этот адрес чтобы вернуть управление коду вызывающей функции. Таким образом, после завершения этого перехода управление вернётся в вызывающую функцию на команду, следующую за командой `CALL g`.

Заметим, что если произойдёт сбой системы, а после повторного запуска системы мы перезапустим программу, операционная система заново выделит процессу память, в которую будет загружен код этого процесса. При этом операционная система может загрузить код процесса в другое место виртуальной памяти, т.е. не по тем же самым адресам, где код был расположен до сбоя. По этой причине записанные в стеке адреса возврата будут бессмысленны: память, на которую они указывают, может больше не содержать кода вообще. В лучшем случае это будет просто недоступная для чтения память, а в худшем случае она будет содержать произвольные данные, исполнение которых как кода вызовёт произвольные ошибки в ходе исполнения.

Следовательно, расположенные на стеке адреса возврата не могут помочь нам понять после возобновления работы системы, какие функции исполнялись в момент сбоя.

Для восстановления после сбоя критически необходимо знать, какие функции исполнялись в момент сбоя. И, так как стек операционной системы не может, по описанным выше причинам, предоставлять необходимую для восстановления информацию, будем поддерживать персистетный стек самостоятельно.

Для каждого потока, имеющего доступ к NVRAM, будем поддерживать персистентный стек особым образом.

Персистентный стек доступен только одному потоку и выделяется до создания этого потока. Стек имеет константный размер и не перевыделяется при переполнении. Всё это делает персистентный стек очень похожим на обычный стек операционной системы. Отличие состоит, во-первых, в том, что персистентный стек хранится в NVRAM, а во-вторых, в том, как этот стек поддерживается.

Персистентный стек состоит из персистентных стековых фреймов — по одному фрейму на каждую функцию, в которой происходит доступ к NVRAM (на чтение или запись). Каждая такая функция должна иметь `recover`-версию, которая получает те же аргументы, что и исходная функция, и должна завершить операцию в случае, если во время её исполнения или исполнения исходной функции произойдёт сбой.

Функции, в коде которых не происходит доступа к NVRAM, а происходит доступ только к локальным данным потока, хранящимся в энергозависимой памяти, не важны для алгоритма. Исполнение этих функций никак не меняет NVRAM, после сбоя результаты их запуска и частичного исполнения не будут видимы. Поэтому для таких функций не нужны `recover`-версии и при вызове таких функций мы не будем создавать персистентные стековые фреймы. Не умаляя общности, можно считать, что тела таких функций подставляются на место их вызова.

Каждый фрейм, кроме последнего, завершается однобайтным маркером конца фрейма (этот маркер имеет вид `0x0`). Последний фрейм завершается однобайтным маркером конца стека (этот маркер имеет вид `0x1`). После конца стека могут идти произвольные данные, эти данные никогда не нужно читать и интерпретировать (см. Рисунок 6).

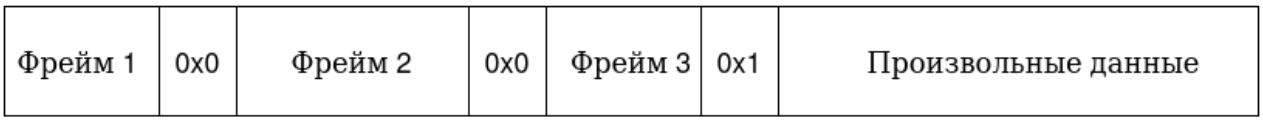

Рисунок 6 – Структура персистентного стека

2.4. Персистентный стековый фрейм

Каждый персистентный стековый фрейм соответствует вызову некой функции `Op`. Фрейм содержит информацию, которая поможет при возобновлении работы системы определить адреса `Op` и `Op.Recover`, размер аргументов функции в байтах, сами аргументы, сериализованные в массив байт, место

под ответ и в последнем байте фрейма маркер конца фрейма или маркер конца стека.

По причинам, объяснённым в секции 2.3, в персистентном фрейме нельзя хранить адреса `Op` и `Op.Recover`.

Для решения этой проблемы можно хранить на стеке не адрес функции, а её уникальный идентификатор, который, в отличие от адреса функции, не будет меняться от запуска к запуску. Например, в качестве идентификатора функции можно использовать имя этой функции (в таком случае требуется чтобы в системе не существовало двух функций с одинаковым именем).

В таком случае нужно уметь сопоставлять идентификатору функции адрес самой функции и её `recover`-версии. Для решения этой проблемы можно хранить в системе глобальную хеш-таблицу, сопоставляющую имени функции два адреса — самой функции и её `recover`-версии. В таком случае при каждом запуске системы необходимо заполнить эту хеш-таблицу, занеся в неё имена и адреса всех имеющих доступ к NVRAM функций, которые будут использоваться в ходе работы. Эта хеш-таблица должна храниться в энергозависимой памяти и заполняться каждый раз при перезапуске процесса, так как с каждым новым стартом системы код функций может менять своё расположение в памяти.

Так как для восстановления системы после сбоя необходимо выполнить `recover`-версии всех операций, которые исполнялись в момент сбоя, на стеке необходимо помимо идентификатора функции (чтобы знать, какую конкретно функцию исполнять) сохранять и её аргументы (чтобы знать, какие аргументы передавать `recover`-функции при запуске).

Будем хранить фреймы на стеке так, чтобы начало каждого фрейма было выровнено по размеру кэш-линии на используемой архитектуре. Смысл этого принципа станет понятен в секции 2.9.

Будем хранить на стеке фреймы со следующей структурой (см. Рисунок 7):

— $0 \leqslant n <$ `CACHE_LINE_SIZE` (где `CACHE_LINE_SIZE` это размер кэш-линии на данной архитектуре в байтах) неиспользуемых байт. Это поле не содержит никакой полезной информации и предназначено для того, чтобы адрес начала фрейма был кратен `CACHE_LINE_SIZE`.

— Поле для возврата ответа. Размер этого поля равен восьми байтам. Его роль будет описана в секции 2.9.
— Длина имени функции в байтах. Это поле имеет тип `uint16_t` и занимает два байта.
— Имя функции, являющееся её уникальным идентификатором. Так как имя функции должно быть уникальным, в системе не должно существовать двух функций с одинаковым именем. Длина этого поля определяется предыдущим числом.
— Длина аргументов функции, сериализованных в массив байт, в байтах. Это поле имеет тип `uint16_t` и занимает два байта.
— Аргументы функции, сериализованные в массив байт. Длина этого поля (количество байт в массиве) определяется предыдущим числом. Пользователь должен самостоятельно выполнять сериализацию/десериализацию аргументов в/из массива байт.
— Маркер конца стека или конца фрейма — один байт.

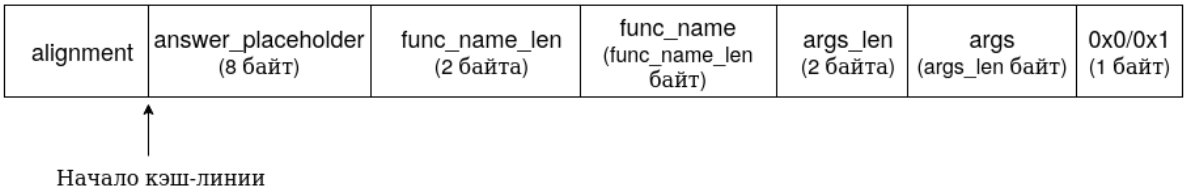

Рисунок 7 – Структура стекового фрейма

Маркер конца фрейма (`0x0`) или стека (`0x1`) позволяет нам понять, был ли прочитанный из персистентного стека фрейм последним или дальше на стеке расположен ещё хотя бы один стековый фрейм.

Таким образом, размер используемой части фрейма равен $8 + 2 + \texttt{func\_name\_len} + 2 + \texttt{args\_len} + 1 = 13 + \texttt{func\_name\_len} + \texttt{args\_len}$ байт. Пустые байты в начале фрейма не используются и нужны только для того, чтобы используемая части фрейма была выровнена по размеру кэш-линии.

2.5. Обновление персистетного стека

Персистентный стек нужно обновлять при вызове новой функции (в этом случае, нужно добавить на вершину стека новый стековый фрейм) и при выходе из функции (в таком случае нужно убрать один стековый фрейм с вершины стека).

Будем называть последним фреймом тот фрейм, который является последним на момент до начала записи нового стекового фрейма. Например, в примере ниже (см. Рисунок 9), фрейм 3 — это новый фрейм, а фрейм 2 — последний фрейм.

Перед входом в функцию будем делать следующие действия (см. Рисунок 8):

Рисунок 8 – До вызова функции

— После маркера конца стека пишем новый стековый фрейм, завершая его маркером конца стека (см. Рисунок 9). Новый стековый фрейм располагается после маркера конца стека, которым завершается последний фрейм, поэтому он пока не считается стековым фреймом.

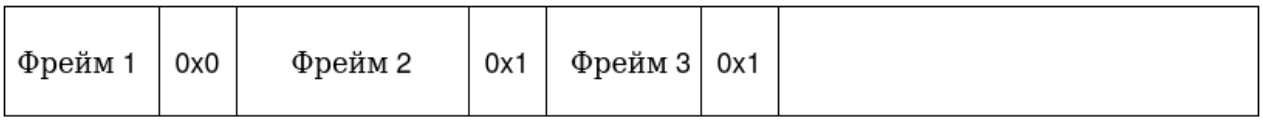

Рисунок 9 – После добавления нового стекового фрейма после конца стека

— У последнего стекового фрейма меняем маркер конца стека на маркер конца фрейма (см. Рисунок 10). При этом последний фрейм становится предпоследним, а новый фрейм становится последним, тем самым происходит перемещение конца стека вперёд.

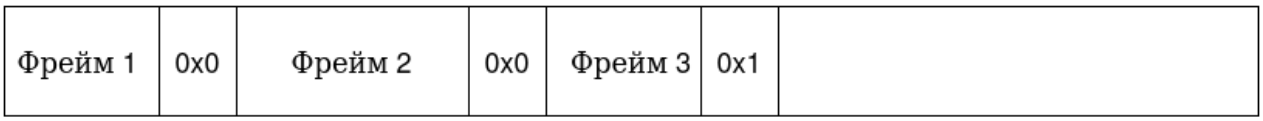

Рисунок 10 – После перемещения конца стека вперёд

Перед выходом из функции будем делать следующие действия (см. Рисунок 11):

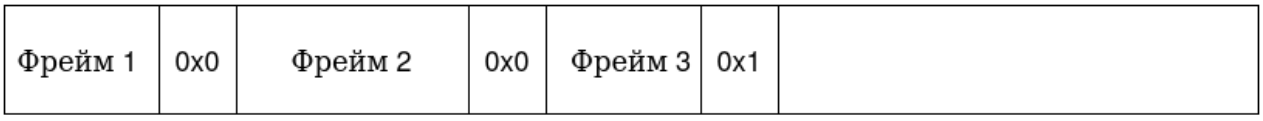

Рисунок 11 – До выхода из функции

— У предпоследнего стекового фрейма меняем маркер конца фрейма на маркер конца стека (см. Рисунок 12). При этом предпоследний стековый фрейм становится последним, а последний стековый фрейм становится набором произвольных данных, расположенных после конца стека, то есть происходит перемещение конца стека назад.

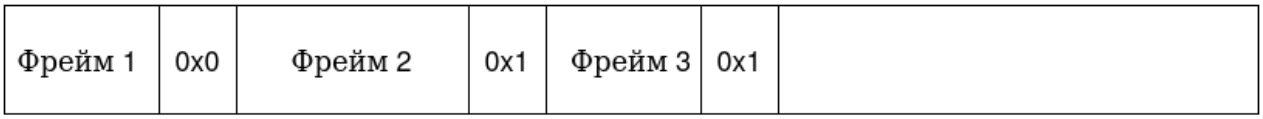

Рисунок 12 – После перемещения конца стека назад

В вышеприведённом алгоритме при выходе из функции необходимо у предпоследнего стекового фрейма заменить маркер конца фрейма на маркер конца стека. Из этого следует, что если предпоследнего фрейма не существует (то есть фрейм, который мы собираемся убрать со стека, является единственным фреймом на стеке), убрать текущий фрейм со стека невозможно. Поэтому первый фрейм, хранимый на стеке, никогда не должен удаляться со стека. Есть два способа добиться этого:

— Считать, что первый фрейм, хранимый на стеке, является фиктивным и не соответствует никакой функции. Этот фиктивный фрейм будет добавляться на стек до начала работы системы и никогда не будет удаляться с вершины стека.

— Потребовать, чтобы первая функция, которая будет запущена в каждом потоке, никогда не завершалась. Единственным способом прервать работу этой функции будет сбой системы. Так как первая вызванная в каждом потоке функция никогда не завершается, то и убирать первый стековый фрейм будет не нужно. В дальнейших главах будет рассмотрена архитектура системы и будет показано, что данное ограничение не является недостатком дизайна, а наоборот, отвечает всем накладываемым на систему требованиям.

2.6. Атомарность входа и выхода

Заметим, что стековый фрейм может не влезть в одну кэш-линию (например потому, что функция получает массив аргументов длиной больше, чем размер кэш-линии). Так как на системах с NVRAM запись в NVRAM несколь-

ких кэш-линий неатомарна, то и запись такого фрейма на стек не будет атомарна.

Следовательно, в ходе записи нового стекового фрейма может произойти сбой, который позволит записать только часть нового стекового фрейма, но не позволит записать остаток фрейма. Например, если сбой произойдёт после записи первой кэш-линии, но до записи всех остальных, новый персистентный стековый фрейм будет записан не полностью.

Заметим, что в процессе записи мы сначала пишем на стек новый стековый фрейм, и только после того, как запись нового фрейма завершилась успешно, мы перемещаем конец стека вперёд (то есть заменяем у последнего стекового фрейма маркер конца стека на маркер конца фрейма). В таком случае, заметим, что вышеописанный сбой может произойти только до перемещения конца стека вперёд, то есть до того, как мы поменяем у последнего фрейма маркер конца стека на маркер конца фрейма. Из этого следует, что не полностью записанный стековый фрейм будет располагаться за концом стека и вообще не будет интерпретироваться как стековый фрейм. То есть в случае такого сбоя мы считаем, что сбой произошёл до входа в функцию и функция вообще не была вызвана в этом потоке. (см. Рисунок 13)

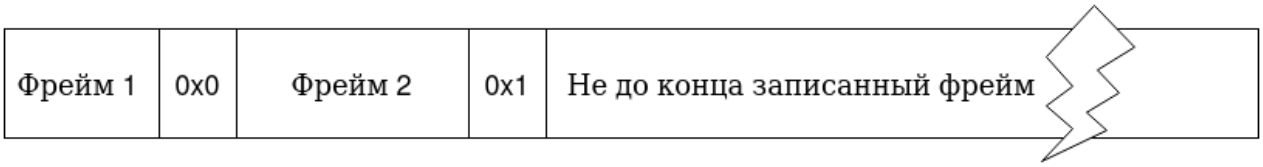

Рисунок 13 – Фрейм, при записи которого произошёл сбой, располагается за концом стека

Заметим также, что перемещение конца стека вперёд сводится к записи в NVRAM одного байта (у последнего фрейма маркер конца стека заменяется на маркер конца фрейма). Точно так же перемещение конца стека назад аналогично сводится к записи в NVRAM одного байта (у предпоследнего фрейма маркер конца фрейма заменяется на маркер конца стека). А так как один байт всегда лежит строго в одной кэш-линии и никогда не пересекает её границ, перемещение конца стека сводится к сбросу в NVRAM единственной кэш-линии (в которой лежит изменённый байт), всегда выполняется атомарно, следовательно, в процессе перемещения конца стека сбой произойти не может.

Если же сбой происходит после перемещения конца стека вперёд/назад, но до реального исполнения процессором команды `CALL`/`RET`, считаем, что вход/выход в/из функции был успешно выполнен и команда `CALL`/`RET` была успешно исполнена. Действительно, исполнение команд `CALL` и `RET` не приводят к записи данных в NVRAM, поэтому после возобновления работы системы ситуация, при которой они были выполнены (и сбой произошёл сразу после выполнения одной из этих команд процессором), неотличима от ситуации, при которой они не были выполнены (то есть сбой произошёл сразу после перемещения конца стека, до исполнения процессором одной из этих команд). В таком случае, будем всегда считать, что вызов функции и возврат из неё происходят в момент перемещения конца стека.

Заметим, что для корректности поддержания стека мы требуем от железа даже не атомарности записи кэш-линии, а всего лишь атомарности записи одного байта (чтобы перемещение конца стека происходило атомарно). Это позволяет эмулировать процесс запуска алгоритмов для NVRAM на более дешёвом железе (например, на жёстких дисках). Современные жёсткие диски могут не гарантировать атомарности записи кэш-линии, зато гарантируют атомарность записи одного байта.

Чтобы вышеприведённые рассуждения были корректны, ни одна запись в персистентный стек не должна быть отложена из-за попадания в кэш. Поэтому сразу после записи любых данных на стек необходимо производить сброс кэш-линии, в которую попала записанные данные, в NVRAM. В противном случае возможна одна из следующих ситуаций:

— При перемещении конца стека вперёд мы пишем маркер конца фрейма вместо маркера конца стека в последний фрейм. Представим, что этот маркер попал в кэш и не записался в NVRAM (см. Рисунок 14). Теперь мы вызываем функцию, соответствующую новому стековому фрейму, и эта функция пишет что-то в NVRAM, причём запись попадает туда сразу, минуя кэш (например, в коде функции записанные данные явно сбрасываются в NVRAM из кэша). Если в этот момент (то есть до того, как маркер конца фрейма будет сброшен в NVRAM) произойдёт сбой, то после возобновления работы системы новый стековый фрейм будет располагаться за маркером конца стека, и, следовательно, мы будем считать, что не вызвали соответствующую ему функцию. Следова-

тельно, `recover`-версия этой функции не будет вызвана. Несмотря на это, функция успела внести какие-то изменения в NVRAM и, не вызвав её `recover`-версию, мы неправильно проведём восстановление системы после сбоя.

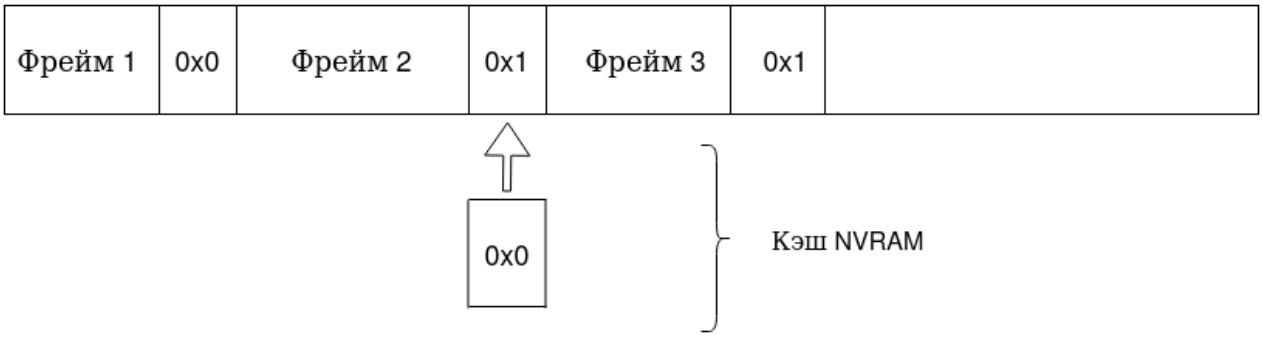

Рисунок 14 – Функция, соответствующая фрейму 3, была вызвана и успела внести изменения в NVRAM, но соответствующий ей фрейм расположен после конца стека, recover-функция не будет вызвана

— Представим себе, что записанные в новый стековый фрейм аргументы вызываемой функции попали в кэш и не были сброшены в NVRAM. Теперь мы вызываем функцию, соответствующую новому стековому фрейму, и эта функция пишет что-то в NVRAM, причём запись попадает туда сразу, минуя кэш (например, в коде функции записанные данные явно сбрасываются в NVRAM из кэша). Если в этот момент (то есть до того, как аргументы вызванной функции будет сброшены в NVRAM) произойдёт сбой системы, после восстановления мы не будем знать, с какими аргументами была вызвана функция и, следовательно, не сможем вызвать `recover`-версию этой функции с теми же аргументами. Следовательно, мы не сможем корректно восстановить систему после сбоя.

Заметим, что в начале каждого стекового фрейма расположены $0 \leqslant n < \texttt{CACHE\_LINE\_SIZE}$ неиспользуемых байт, записанное в которых значение нам не важно. Мы никогда не читаем и не интерпретируем эти байты, они нужны только для выравнивания поля для сохранения ответа по размеру кэш-линии. Тогда при сбросе нового стекового фрейма в NVRAM нужно сбрасывать в NVRAM не весь новый фрейм, а только его используемую часть. В таком случае, сбрасывается в NVRAM всегда строго

$13 + \texttt{func\_name\_len} + \texttt{args\_len}$ байт, выровненных по кэш-линии. В таком случае, если $13 + \texttt{func\_name\_len} + \texttt{args\_len} \leqslant \texttt{CACHE\_LINE\_SIZE}$, для сброса нового стекового фрейма в NVRAM понадобится всего одна операция сброса кэш-линии.

2.7. Работа с указателями

При работе с указателями на NVRAM мы сталкиваемся с проблемами, аналогичными тем, с которыми мы столкнулись при работе с указателями на функции.

Мы эмулируем NVRAM с помощью жёстких дисков. Эмуляция происходит следующим образом: сначала мы создаём файл нужного размера на жёстком диске, потом открываем его с помощью системного вызова `open(2)`, получая файловый дескриптор, соответствующий открытому файлу. Открыв файл, мы отображаем его в память с помощью системного вызова `mmap(2)`, тем самым получив возможность работать с байтоадресуемой памятью, запись в которую приводит к записи в постоянное хранилище [15].

Как и в случае с NVRAM, запись в память, выделенную с помощью `mmap` не всегда приводит к немедленной записи в постоянное хранилище. Записанные данные могут остаться в буфере в ядре операционной системы, играющем для отображаемого в память файла ту же роль, что и кэш для NVRAM (то есть при сбое системы даные из буфера пропадают, не попадая в энергонезависимое хранилище). Так же, как и в случае со сбросом кэша в NVRAM, можно с помощью системного вызова `msync` явно потребовать сброса записанных данных из буфера в ядре операционной системы в энергонезависимое хранилище [17].

Будем хранить персистетный стек каждого процесса в собственном файле, и с помощью отдельного файла мы будем эмулировать кучу NVRAM. Таким образом, если в системе есть `N` потоков, мы будем использовать `N + 1` файл для эмуляции NVRAM: `N` для эмуляции стеков потоков и один для эмуляции кучи NVRAM.

Представим теперь ситуацию, при которой мы передали вызванной функции `f(void*)` указатель на участок в куче NVRAM в качестве аргумента. Так как аргументы вызываемой функции записываются на персистетный стек, то и передаваемый указатель на NVRAM также будет записан на стек. Пусть в момент работы функции `f(void*)` произошёл сбой системы, после

чего работа системы была возобновлена, а процесс перезапущен. После перезапуска процесса мы вновь открываем файл, с помощью которого мы эмулируем кучу NVRAM, и отображаем его в память. Операционная система могла отобразить файл в другое место виртуальной памяти, отличное от того, куда он был отображён до сбоя. Это приведёт к тому, что лежащий на персистетном стеке адрес может более не указывать на кучу NVRAM. Следовательно, указатель, который мы передадим `f_recover(void*)` не будет более указывать в кучу NVRAM, поэтому нам не удастся корректно восстановить систему после сбоя.

Замена эмуляции NVRAM с помощью отображаемых в память файлов на настоящую NVRAM также может не решить эту проблему. Скорее всего, NVRAM будет представлять из себя отдельное устройство, отображаемое в произвольное место виртуальной памяти с помощью системного вызова `mmap(2)` или подобного ему (то есть вместо использования механизма отображения файла в память мы будем использовать механизм отображения устройства в память), что порождает те же проблемы, что и при эмуляции NVRAM с использованием жёстких дисков.

Поэтому везде при работе с NVRAM будем хранить не абсолютные адреса объектов в NVRAM, а смещения от адреса начала отображения NVRAM в виртуальное адресное пространство. Адрес начала отображения NVRAM в виртуальное адресное пространство должен храниться в глобальной переменной `PMEM_START_ADDRESS` в энергозависимой памяти и инициализироваться каждый раз при старте системы. Например, при эмуляции NVRAM с использованием отображаемых в память файлов `PMEM_START_ADDRESS` будет инициализироваться результатом вызова `mmap`, а при работе с настоящей NVRAM — результатом вызова специальной функции, отображающей NVRAM в виртуальное адресное пространство.

Заметим, что смещения не зависят от конкретного значения `PMEM_START_ADDRESS`, поэтому их можно сохранять в энергонезависимой памяти и использовать как корректное смещение даже после перезапуска процесса.

В ходе исполнения программы смещения можно преобразовывать в абсолютные адреса (по которым мы и будем обращаться к NVRAM). Например, имея смещение от адреса начала отображения NVRAM в виртуальное адрес-

ное пространство, можно легко получить абсолютный адрес, прибавив смещение к PMEM_START_ADDRESS.

В обратную сторону, смещение же легко получается из абсолютного адреса с помощью вычитания из абсолютного адреса PMEM_START_ADDRESS.

2.8. Возвращаемое значение

Традиционно, на архитектуре x86 возвращаемые из функций значения передаются вызывающей стороне либо в регистре общего назначения EAX (если возвращаемое значение является целым числом или указателем), либо в регистрах арифметического сопроцессора (если возвращаемое значение является числом с плавающей точкой). Например, именно таковы правила возврата значений в конвенции вызова cdecl, одной из самых распространённых конвенций вызова, используемых на архитектуре x86 [29].

Тем не менее, при работе с NVRAM возвращаемое значение функции нельзя возвращать в регистрах процессора, так как регистры процессора энергозависимы. Представим, что функция Op завершилась и записала возвращаемое значение в регистр EAX. После этого произошло перемещение конца стека назад (вследствие чего стековый фрейм, соответствующий функции Op, был выброшен с вершины стека), после чего была исполнена команда RET, и в этот момент произошёл сбой.

С одной стороны, сбой произошёл до того, как вызывающая сторона успела сохранить значение из энергозависимого регистра EAX в NVRAM, поэтому после возобновления работы системы возвращаемое значение будет потеряно. С другой стороны, сбой произошёл после перемещения конца стека назад, поэтому при анализе стека после возобновления работы системы операция Op будет считаться корректно завершившейся, и Op.Recover не будет вызвана. Вследствие этого, возвращаемое значение операции Op будет потеряно навсегда и восстановить его не будет возможности. Следовательно, мы не сможем корректно восстановить систему после сбоя.

Для борьбы с проблемой такого рода необходимо записывать возвращаемое значение в NVRAM до того, как произойдёт перемещение конца стека назад.

Делать это можно следующим образом:

— Перед вызовом функции вызывающая сторона выделяет память в NVRAM, доступ к которой имеет только один процесс.

— Вызывающая сторона передаёт указатель на эту память вызываемой функции в качестве параметра.
— Перед завершением вызываемая функция сохраняет по этому адресу ответ и производит сброс ответа из кэша в NVRAM.
— Только после завершения сброса кэшей происходит перемещение конца стека назад.

Важно иметь в виду, что перемещение конца стека назад происходит не просто после записи возвращаемого значения в NVRAM, а после сброса возвращаемого значения из кэша в NVRAM. В противном случае, мы столкнёмся с проблемой, аналогичной той, с которой мы столкнулись, возвращая значение в энергозависимых регистрах.

Представим, что функция `Op` завершилась и записала возвращаемое значение в NVRAM, но при этом не сбросила кэши и возвращаемое значение осталось в кэше, не попав в NVRAM. После этого произошло перемещение конца стека назад (вследствие чего стековый фрейм, соответствующий функции `Op`, был выброшен с вершины стека), после чего была исполнена команда `RET`, и в этот момент произошёл сбой.

С одной стороны, сбой произошёл до того, как возвращаемое значение попало из кэша в NVRAM, поэтому после возобновления работы системы возвращаемое значение будет потеряно. С другой стороны, сбой произошёл после перемещения конца стека назад, поэтому при анализе стека после возобновления работы системы операция `Op` будет считаться корректно завершившейся, и `Op.Recover` не будет вызвана. Вследствие этого, возвращаемое значение операции `Op` будет потеряно навсегда и восстановить его не будет возможности. Следовательно, мы не сможем корректно восстановить систему после сбоя.

2.9. Возврат значения на стеке

Кроме описанного выше способа возврата значения, существуют и другие. Например, значение можно возвращать на стеке.

Будем в начале каждого стекового фрейма, в качестве первого поля фрейма, сохранять восемь байт под ответ. Если функция `G`, вызванная из функции `F` хочет записать ответ на стек NVRAM, она должна записать ответ в начало предыдущего стекового фрейма – это будет стековый фрейм, соответствующий функции `F` (см. Рисунок 15) – и сбросить возвращаемое значение из

кэша в NVRAM (в предыдущей главе было описано, почему функция должна сбрасывать собственное возвращаемое значение из кэша в NVRAM до того, как произойдёт перемещение конца стека назад).

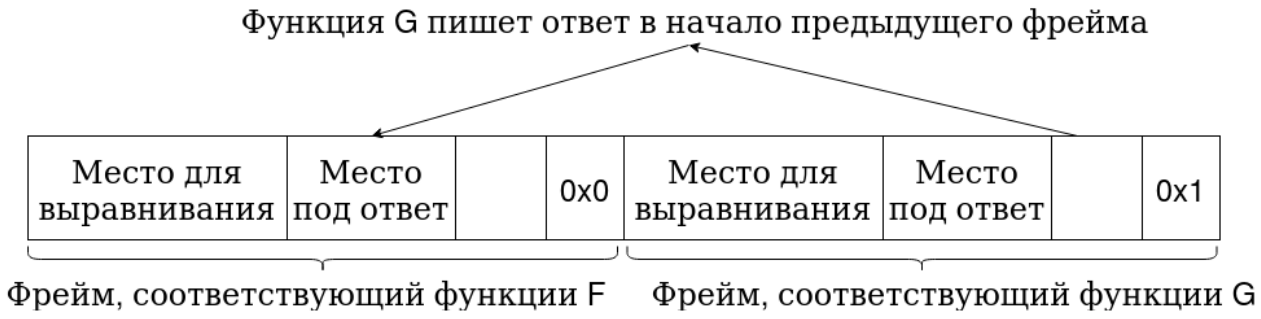

Рисунок 15 – Запись возвращаемого значения в начало предыдущего фрейма

Функция `F` может получить ответ функции `G` после её завершения, для этого достаточно прочитать первые восемь байт собственного стекового фрейма.

Разумеется, если функция `F` после завершения работы функции `G` вызовет возвращающую ответ функцию `H`, то после завершения работы функции `H` в начале стекового фрейма, соответствующего функции `F`, будет записан ответ функции `H`, а ответ функции `G` будет стёрт. Если функции `F` необходимо сохранить возвращаемое значение функции `G`, она должна сделать это явно, перед вызовом функции `H` перенеся возвращаемое значение из начала собственного стекового фрейма в заранее выделенный в куче NVRAM участок памяти.

Так как мы хотим, чтобы запись возвращаемого значения выполнялась атомарно, мы выполним выравнивание адреса начала возвращаемого значения в каждом стековом фрейме по размеру кэш-линии. Именно для этого в начале каждого фрейма мы пропускаем $0 \leqslant n < \texttt{CACHE\_LINE\_SIZE}$ неиспользуемых байт. Так как адрес начала возвращаемого значение выровнен по размеру кэш-линии (см. Рисунок 16), запись возвращаемого значения сводится к записи восьми байт, лежащих строго в одной кэш-линии и не пересекающих её границ (разумеется, для этого размер кэш-линии должен быть не меньше восьми байт — к счастью, на архитектуре `x86` размер кэш-линии равен 64 байтам), поэтому в системах с NVRAM такая операция выполняется атомарно.

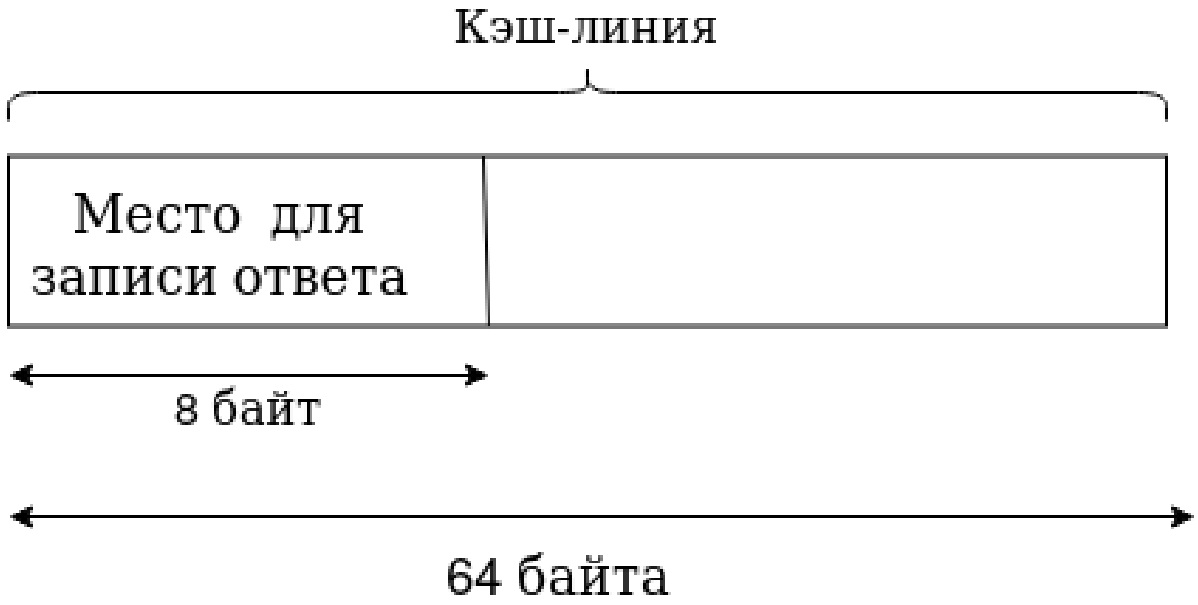

Рисунок 16 – Положение места для записи возвращаемого значения в кэш-линии архитектуры x86

В вышеприведённом алгоритме функция пишет возвращаемое значение в начало предыдущего стекового фрейма. Из этого следует, что если предыдущего фрейма не существует (то есть фрейм, соответствующий исполняемой в данный момент функции, является единственным фреймом на стеке), вернуть значение из исполняющейся в данный момент функции невозможно. Как и в случае с запретом на удаление первого фрейма со стека, есть два способа добиться того, чтобы функция, соответствующая первому фрейму, никогда не возвращала ответ:

— Считать, что первый фрейм, хранимый на стеке, является фиктивным и не соответствует никакой функции. Так как первый фрейм не соответствует никакой функции, то и вернуть значение функция, соответствующая ему, не может.
— Потребовать, чтобы первая функция, которая будет запущена в каждом потоке, никогда не завершалась. Единственным способом прервать работу этой функции будет сбой системы. Так как первая вызванная в каждом потоке функция никогда не завершается, то и возвращать из неё значение не нужно. В дальнейших главах будет рассмотрена архитектура системы и будет показано, что данное ограничение не является недостатком дизайна, а наоборот, отвечает всем накладываемым на систему требованиям.

К сожалению, жёсткие диски не предоставляют гарантию атомарности записи кэш-линии, поэтому на системах, использующих жёсткие диски в качестве энергонезависимого хранилища, может произойти сбой в процессе записи ответа, в результате которого ответ будет записан не целиком. Пользовательский код самостоятельно должен учитывать возможность такого сбоя. Например, для борьбы с такими сбоями можно использовать следующий протокол:

- Перед записью нового фрейма на стек запишем в каждый из восьми байт места под ответ текущего стекового фрейма значение по умолчанию (например, `0xFF`). Подразумевается, что ни один байт ответа не может быть равен этому значению.
- Вызываемая функция пишет ответ на стек обычным образом.
- `recover`-версия вызываемой функции первым делом читает ответ с предыдущего фрейма.
- Дальше возможны три варианта:
  - Все байты ответа содержат значение по умолчанию. Следовательно, сбой произошёл до того, как ответ был записан на персистентный стек, необходимо провести восстановление.
  - Ни один из байт ответа не содержит значения по умолчанию. Следовательно, сбой произошёл уже после того, как ответ был записан на персистентный стек. Восстановления производить не нужно, `recover`-функция тут же может завершить свою работу.
  - Только часть байт ответа содержит значения по умолчанию. Следовательно, сбой произошёл в процессе записи ответа на персистентный стек, необходимо провести восстановление.

Заметим, что если ответ функции занимает один байт (например, ответ функции `CAS` занимает всего один байт) и ответ пишется в первый байт персистентного фрейма, то сбой в процессе записи ответа невозможен даже при эмуляции алгоритма на жёстких дисках (так как жёсткие диски предоставляют гарантию атомарности записи одного байта).

2.10. Взаимодействие с пользователем

Для работы с системой запуска программ для NVRAM от пользователя требуется выполнять следующие действия:

— Для всех функций, имеющих доступ к NVRAM, предоставлять `recover`-версию. Эта версия должна принимать те же аргументы, что и обычная функция
— Все такие функции должны принимать аргументы, сериализованные в массив байт и иметь сигнатуру
  `void func(const uint8_t* args)`
  `void func_recover(const uint8_t* args)`
  При вызове функции пользователь сам должен сериализовать передаваемые ей аргументы в массив байт, а также десериализовывать аргументы, получаемые пользовательской функцией, из массива байт.
— Вызывать функции, работающие с NVRAM, нельзя напрямую. Нужно делать это через функцию `void do_call(std::string const func_name, std::vector<uint8_t> const args)`. Эта функция до вызова пользовательской функции создаёт стековый фрейм и производит перемещение конца стека вперёд, а после завершения пользовательской функции производит перемещение конца стека назад.
— Перед завершением функции нужно самостоятельно сохранять ответ в NVRAM. Сохранение ответа на стек производится с помощью функции `void write_answer(std::vector<uint8_t> const answer)`, сохранение в кучу NVRAM пользователь должен делать самостоятельно, используя описанный в секции 2.8 протокол. Для чтения ответа с текущего стекового фрейма (то есть ответа, записанного функцией, только что завершившей свою работу) можно использовать функцию `std::vector<uint8_t> read_answer(uint8_t size)`, а для чтения ответа с предыдущего стекового фрейма (то есть ответа, записанного функцией, исполняющейся в данный момент) можно использовать функцию `std::vector<uint8_t> read_current_answer(uint8_t size)`.
— Перед началом работы регистрировать в глобальной хеш-таблице все используемые функции, запоминая по имени функции адрес самой функции и адрес её `recover`-версии. В дальнейшем можно использовать только заранее зарегистрированные функции.
— Самостоятельно передавать указатели на кучу NVRAM как смещения от адреса начала отображения NVRAM в виртуальное адресное про-

странство. При необходимости получить данные по указателю необходимо самостоятельно прибавлять смещение к `PMEM_START_ADDRESS`, получая абсолютный адрес; при необходимости получить смещение из абсолютного адреса — вычитать из абсолютного адреса `PMEM_START_ADDRESS`. Пользователь должен самостоятельно заботиться о том, чтобы нигде в NVRAM не сохранялись указатели на кучу NVRAM вместо смещений.

2.11. Архитектура системы

Система состоит из главного потока и `N` рабочих потоков.

Главный поток может быть запущен в двух режимах: в штатном режиме и в режиме восстановления.

При запуске в штатном режиме главный поток:

— Инициализирует кучу NVRAM: создаёт на жёстком диске файл требуемого размера, с помощью которого будет производиться эмуляция кучи, открывает его и отображает в память, сохраняет в переменной `PMEM_START_ADDRESS` адрес начала отображения файла в виртуальное адресное пространство.

— Если это первый запуск, главный поток инициализирует аллокатор памяти (вопрос выделения памяти в куче NVRAM будет рассмотрен в главе 3).

— Регистрирует в глобальной хеш-таблице имена и адреса всех функций (и их `recover`-версий), которые будут использоваться в ходе исполнения.

— Создаёт `N` персистетных стеков. При эмуляции NVRAM с помощью жёстких дисков аллоцировать можно как каждый стек в собственном файле, так и все стеки в единственном файле (в таком случае каждый рабочий поток получит указатель на различные смещения в одном и том же файле). При реализации системы я решил аллоцировать каждый стек в собственном файле, чтобы ошибки при реализации алгоритма работы с персистентным стеком одного рабочего потока не приводили к повреждению персистентных стеков других рабочих потоков.

— Запускает `N` рабочих потоков и выдаёт каждому рабочему потоку указатель на начало его персистетного стека.

— Добавляет задачи, которые будут исполнять рабочие потоки, в очередь задач. В качестве очереди задач использовалась обычная очередь из

стандартной библиотеки языка C++, защищённая грубой блокировкой для корректной работы в многопоточном окружении. Также при реализации очереди использовались мониторы Хоара [16] для того, чтобы рабочий поток мог пассивно ожидать появления задач в очереди при попытке взять задачу из пустой очереди, а главный поток мог оповестить рабочий поток о том, что в очереди появилась новая задача при добавлении задачи в очередь. Очередь не является персистентной и располагается в энергозависимой памяти.

Запуску в режиме восстановления предшествует сбой системы. При запуске в режиме восстановления главный поток:

— Инициализирует кучу NVRAM: открывает файл и отображает его в память, сохраняет в переменной PMEM_START_ADDRESS адрес начала отображения файла в виртуальное адресное пространство. Аллокатор при этом не инициализируется, так как он был инициализирован при запуске в штатном режиме.
— Регистрирует в глобальной хеш-таблице имена и адреса всех функций (и их recover-версий), которые будут использоваться в ходе исполнения.
— Открывает файлы, соответствующие персистентным стекам рабочих потоков, исполнявшихся во время сбоя и отображает их в память.
— Запускает N восстанавливающих потоков и выдаёт каждому из восстанавливающих потоков указатель на соответствующий персистетный стек, открытый на предыдущем шаге. Так как каждому восстанавливающему потоку выдаётся ровно один стек рабочего потока, число восстанавливающих потоков в системе должно быть равно числу рабочих потоков.
— Дожидается завершения всех восстанавливающих потоков и после этого перезапускается в штатном режиме. Заметим, что этот запуск в штатном режиме не будет являться первым запуском в штатном режиме. В частности, аллокатор памяти уже будет инициализирован и не нужно будет инициализировать его повторно.

Рабочие потоки при этом разделяют очередь задач и исполняют эти задачи.

Задача содержит описание того, что необходимо сделать (например, там может быть указано, что необходимо выполнить операцию `CAS` по переменной, лежащей в куче NVRAM по смещению `0x2517` с аргументами `expected = 100, new = 200`) и указатель на выделенную в NVRAM память, куда пишется результат исполнения описанной операции. Эта память инициализируется главным потоком при создании задачи и доступна всего двум потокам: рабочий поток, исполняющий задачу, единственный раз пишет в неё результат исполнения операции, а главный поток читает эту память для того, чтобы узнать результат.

Изначально эта память инициализируется значением по умолчанию, которое не может быть результатом исполнения задачи (например, для `CAS` значением по умолчанию может стать `-1`, а возможными результатами исполнения — `0` и `1`). Сравнивая текущее записанное в переменной значение со значением по умолчанию, главный поток может узнать, закончилось ли выполнение задачи. Если исполнение задачи закончилось, главный поток более не будет обращаться к этой памяти. В противном случае, главный поток обратится к этой памяти ещё раз через какое-то время чтобы ещё раз узнать, завершилось ли исполнение задачи.

Вспомним, что первый фрейм, расположенный на персистетном стеке, не может быть удалён со стека. Следовательно, функция, котрая была вызвана первой в каждом из рабочих потоков, никогда не должна завершаться. Кроме того, вспомним, что ответ функции всегда пишется на предыдущий стековый фрейм, следовательно, первая вызванная в каждом потоке функция не должна возвращать ответ.

Заметим, что эти ограничения удовлетворяют архитектуре системы. Функция, которая вызывается первой в каждом из рабочих потоков, в бесконечном цикле ожидает, используя очередь, заданий от главного потока и исполняет эти задания. Эта функция никогда не завершается нормальным образом, никогда не возвращает ответ, единственным способом завершить работу этой функции является сбой системы.

Заметим, что эта функция никак не взаимодействует с NVRAM. Тем не менее, мы будем создавать для неё стековый фрейм на персистентном стеке (это будет первый фрейм на стеке каждого потока), который мы никогда не будем удалять с персистентного стека. Так как функция никак не взаимодей-

ствиует с NVRAM, её `recover`-версия не производит никаких действий и тут же завершается.

2.12. Восстановление после сбоя

При сбое системы мы для начала устраняем проблемы оборудования, если нужно (например, включаем питание), после чего перезапускаем систему.

Мы запускаем `N` восстанавливающих потоков — по одному восстанавливающему потоку на каждый рабочий поток. Каждому восстанавливающему потоку мы предоставляем ссылку на персистентный стек соответствующего рабочего потока.

После чего каждый восстанавливающий поток проходится по собственному стеку от конца к началу, исполняя `recover`-операцию для каждого персистентного стекового фрейма. По завершении `recover`-операции поток перемещает конец стека назад и повторяет эти действия до тех пор, пока его стек не опустеет.

Таким образом, восстановление происходит параллельно, что позволяет осуществлять потенциально более быстрое восстановление, чем восстановление в один поток.

Если в процессе восстановления произойдёт повторный сбой, то после возобновления работы системы восстановление после повторного сбоя начнётся с того места, где прервалось восстановление в прошлый раз (см. Рисунок 17). Заметим, что при завершении каждой `recover`-операции восстанавливающий поток передвигает конец стека назад, тем самым уменьшая размер стека с каждой выполненной `recover`-операцией. Таким образом, при восстановлении в системе всегда есть прогресс, даже при повторных сбоях.

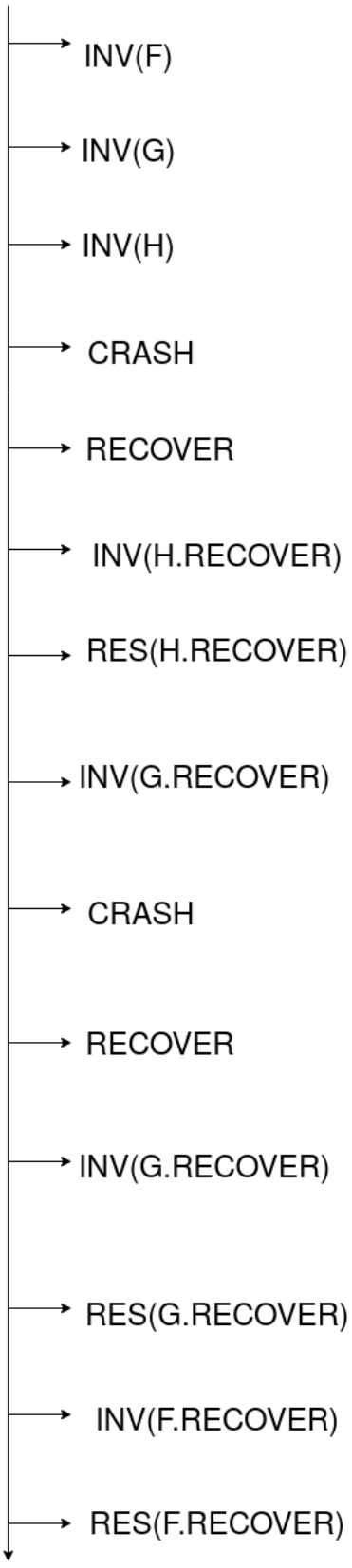

Рисунок 17 – Восстановление системы в условиях повторных сбоев

2.13. Использование алгоритма для запуска программ, удовлетворяющих более слабым условиям корректности

Заметим, что при запуске кода, удовлетворяющего условию корректности *nesting-safe recoverable linearizability* нет разницы между завершением операции `Op` и завершением операции `Op.Recover`. Если некая операция была завершена, то для вызывающего эту операцию кода не должно быть разницы, завершилась она со сбоем или нет.

Однако, существуют условия корректности, в которых эта разница значительна. Рассмотрим, например, условие корректности *durable linearizability*, описанное в секции 1.3. Это условие корректности требует, чтобы после восстановления системы после сбоя был виден эффект всех операций, исполнение которых завершилось до сбоя. Если же операция исполнялась в момент сбоя, то результат её работы может быть как виден, так и не виден.

Существует очевидный способ написать код, удовлетворяющий этому условию корректности: пусть каждой операции `Op`, в коде которой происходит доступ к NVRAM, будет сопоставлена операция `Op.Cleanup`, которая получает те же аргументы, что и исходная операция. В случае сбоя системы `Op.Cleanup` должна откатить обратно все изменения, которые были внесены в NVRAM в ходе исполнения операции `Op`. Тогда `cleanup`-версии всех операций, исполнявшихся в момент сбя, откатят изменения, которые внесли в NVRAM эти операции до сбоя, и после восстановления системы эффект этих операций будет не виден.

В таком случае, очевидно, исполняя `Op.Cleanup` по тем же правилам, по которым мы исполняли `Op.Recover`, мы получим возможность запускать код, удовлетворяющий условию корректности durable linearizability, для этого нам даже не нужно будет менять ни строчки в коде алгоритма запуска и восстановления.

Кроме того, в коде, удовлетворяющем условию корректности *durable linearizability*, после восстановления системы должен быть виден эффект всех операций, исполнение которых завершилось до момента сбоя. Для этого каждая операция `Op` перед завершением должна сбросить в NVRAM все кэш-линии, которые были изменены в ходе исполнения этой операции.

Условие корректности *buffered durable linearizability* отличается от *durable linearizability* только тем, что после восстановления системы эффект

части операций, завершившихся до сбоя, может быть не виден. Но в таком случае объект, операции с которым удовлетворяют условию корректности *buffered durable linearizability*, должен предоставлять операцию `sync()`. Если исполнение этой операции завершилось до сбоя, то после восстановления системы должны быть видимы эффекты всех операций, исполнение которых завершилось до вызова `sync()`. Эффекты исполнения операций, исполнение которых завершилось после вызова `sync()`, но до сбоя, а также тех операций, которые исполнялись в момент сбоя, могут быть не видимы после восстановления системы.

Алгоритм запуска программ, удовлетворяющих этому условию корректности, не отличается от алгоритма запуска программ, удовлетворяющих условию *durable linearizability*. Но, в отличие от *durable linearizability*, операции, удовлетворяющие условию *buffered durable linearizability* не обязаны перед завершением работы производить сброс в NVRAM всех кэш-линий, изменённых в процессе операции. Операция `sync()` должна быть реализована таким образом, что все кэш-линии, изменённые до момента вызова операции `sync()`, перед завершением этой операции должны быть сброшены в NVRAM из кэша.

2.14. Реализация алгоритма

Заметим, что разработанный алгоритм требует от энергонезависимого хранилища гарантию атомарности записи всего одного байта. Жёсткие диски, в отличие от NVRAM, могут не предоставлять гарантию атомарной записи целой кэш-линии, но они гарантируют атомарность записи одного байта. Именно поэтому разработанный алгоритм можно реализовывать не только на системе с использованием NVRAM, но и на системе, использующей жёсткие диски в качестве энергонезависимого хранилища.

При реализации алгоритма в качестве базовового примитива многопоточного исполнения будем использовать потоки, а не процессы, по следующим причинам:

— Все потоки одного процесса разделяют память, вследствие чего при использовании потоков можно, например, с лёгкостью использовать разделяемую очередь заданий, расположенную в общей для всех потоков памяти. Разделение же памяти между процессами потребовало бы использования продвинутых механизмов межпроцессного взаимодействия, в

частности shared memory-mapped files [15], pipes [19], POSIX message queues [21], file locking [6], POSIX named semaphores [22].

— Все потоки одного процесса могут быть с лёгкостью немедленно остановлены с помощью посылания сигнала `SIGKILL` этому процессу. Так как сигнал `SIGKILL`, посланный процессу, не может быть перехвачен и обработан и этот сигнал немедленно завершает процесс, именно с помощью посылания процессу этого сигнала можно эмулировать сбои системы. В противоположность использованию потоков, несколько процессов не могут быть легко остановлены с использованием всего олного сигнала.

Алгоритм был реализован на языке программирования `C++`. Этот язык обладает очень высокой производительностью и богатыми выразительными возможностями, позволяющими реализовывать (и, что тоже очень важно, правильно типизировать) высокоуровневые абстракции, такие как `shared_ptr`, `weak_ptr` и `unique_ptr` (освобождающие программиста от необходимости управлять памятью вручную) или `variant` (позволяющий программисту использовать алгебраические типы). Кроме того, этот язык обладает богатым набором библиотек для решения различных посведневных задач (например, при реализации системы для написания тестов использовалась библиотека `gtest` [7]). Кроме того, этот язык позволяет напрямую использовать системные вызовы операционной системы Linux.

При реализации алгоритма использовалась библиотека `PMDK`, облагчающая работу с энергонезависимой памятью [20].

Выводы по главе 2

В этой главе был разработан алгоритм запуска программ в NVRAM и восстановления системы после сбоя. В ходе проектирования алгоритма был разработан алгоритм обновления персистентного стека, рассмотрены вопросы атомарности входа в функцию и выхода из неё. Алгоритм был разработан с учётом возможности реализации на системе, использующей жёсткие диски в качестве энергонезависимого хранилища.

Была разработана архитектура системы, рассмотрена возможность повторных сбоев в ходе восстановления. Алгоритм разработан с учётом возможности повторных сбоев таким образом, чтобы алгоритм совершал прогресс даже при наличии в системе повторных сбоев.

Кроме того, были рассмотрены вопросы возврата значения из функций и корректной работы с адресами функций и указателями на NVRAM.

Рассмотренный апгоритм может быть использован для запуска программ, удовлетворяющих как условию корректности Nesting-safe recoverable linearizability, так и более слабым критериям корректности.

ГЛАВА 3. ВЫДЕЛЕНИЕ ПАМЯТИ В КУЧЕ NVRAM

3.1. Задача выделения памяти в куче NVRAM

Задача выделения памяти в куче NVRAM состоит из:

— Выделения динамической памяти произвольного размера в куче NVRAM с помощью функции `void* pmem_alloc(size_t size)`, принимающей в качестве параметра длину блока, который необходимо выделить в куче NVRAM (в байтах) и возвращающей указатель на первый байт этого блока.

— Освобождения динамической памяти в куче NVRAM с помощью функции `void pmem_free(void* ptr)`, принимающей в качестве параметра указатель на первый байт блока, выделенного в куче NVRAM с помощью функции `pmem_alloc`. Освобождаемый блок должен быть выделен в куче NVRAM с помощью функции `pmem_alloc`, и на момент вызова функции `pmem_free` этот блок не должен быть освобождён. Несоблюдение одного из этих условий приводит к неопределённому поведению.

Таким образом, задача выделения памяти в куче NVRAM очень похожа на задачу выделения динамической памяти в куче RAM, решаемую, например, аллокатором `malloc` [14].

Подробнее задача выделения памяти в куче NVRAM рассматривается в статье [28].

Заметим, что в описанном в главе 2 алгоритме динамическую память нужно выделять только под ответ каждой задачи, исполняемой рабочими потоками. Если задача имеет ответ фиксированного размера, то можно написать более простой алгоритм, который к тому же не требует от системы гарантии атомарности записи кэш-линии (а только атомарности записи одного байта).

В этой главе рассмотрены два алгоритма выделения динамической памяти в куче NVRAM: первый алгоритм будет способен выделять память произвольного размера, но будет неспособен освобождать память, а также будет требовать от системы гарантии атомарности записи восьми байт. Второй алгоритм будет способен выделять блоки памяти только фиксированного размера, зато будет способен освобождать выделенную память и будет требовать от системы гарантии атомарности записи всего одного байта.

3.2. Первый алгоритм

Будем считать, что в каждый момент времени в куче NVRAM выделен непрерывный отрезок памяти со смещениями в диапазоне [0; CUR_MAX_OFFSET]. Следовательно, CUR_MAX_OFFSET определяет смещение последнего выделенного байта. Как и в предыдущеё главе, смещения будем считать от начала отображения кучи NVRAM (или файла, эмулирующего кучу NVRAM) в виртуальное адресное пространство.

Будем хранить CUR_MAX_OFFSET в начальных восьми байтах кучи NVRAM, то есть в отрезке памяти, имеющем смещения [0; 7] (см. Рисунок 18). Следовательно, изначально CUR_MAX_OFFSET = 7. То есть при инициализации аллокатора необходимо в первые восемь байт кучи NVRAM записать число 7.

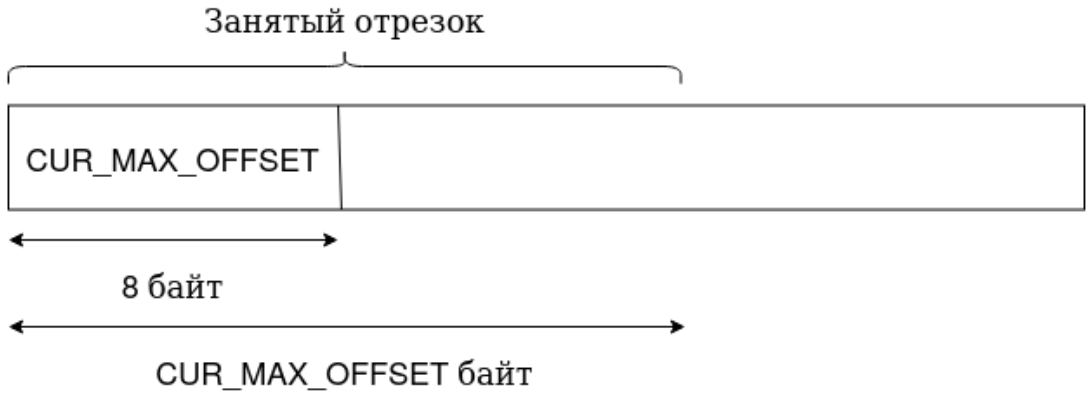

Рисунок 18 – Структура кучи NVRAM в первом алгоритме аллокации

При необходимости выделить блок памяти размером size байт:

— Сохраним в локальную переменную offset текущее значение CUR_MAX_OFFSET, прочитав первые 8 байт кучи NVRAM. После этого шага в локальной переменной offset хранится смещение последнего выделенного в куче NVRAM байта.
— Так как текущий последний выделенный байт имеет смещение offset, после выделения блока памяти размером size байт последний выделенный байт будет иметь смещение offset + size. Следовательно, для выделения памяти запишем в первые 8 байт кучи NVRAM значение offset + size.
— Так как в переменной offset хранится смещение последнего выделенного байта, только что выделенный блок памяти будет начинаться

со смещения `offset + 1`, именно это число и возвращает функция `pmem_alloc`.

— Для того, чтобы предотвратить одновременный доступ нескольких потоков к аллокатору, будем защищать функцию `pmem_alloc` грубой блокировкой, расположенной в энергозависимой памяти.

Освобождение памяти не производится.

Рассмотрим псевдокод этого алгоритма (см. Листинг 1).

Листинг 1 – Псевдокод алгоритма выделения памяти

```
function pmem_alloc(size)
    AllocatorLock.lock()
    uint64_t cur_max_offset
    pmem_read(0, &cur_max_offset, 8)
    next_max_offset = cur_max_offset + size
    pmem_write(0, &next_max_offset, 8)
    pmem_flush(0, 8)
    AllocatorLock.unlock()
    return cur_max_offset + 1
end function

function pmem_free(ptr)
    return
end function
```

Заметим, что первые восемь байт кучи NVRAM выровнены по размеру кэш-линии (так как начало отображения файла или устройства в виртуальное адресное пространство выровнено по размеру страницы операционной системы [15] и, следовательно, выровнено по размеру кэш-линии), следовательно, на NVRAM все первые восемь байт можно записать атомарно. Но для эмуляции NVRAM с помощью жёстких дисков имеет смысл разработать алгоритм, который бы работал корректно при наличии возможности атомарно записать только один байт.

3.3. Второй алгоритм: инварианты, состояние аллокатора и структура кучи NVRAM

Будем требовать от алгоритма выделять только блоки одного и того же размера (по k байт каждый блок).

Разделим все байты на группы по $k+1$ последовательных байт в каждой группе. Первые k байт группы могут быть выделены пользователю, а последний байт группы хранит одно из трёх возможных состояний: `0x0`, `0x1` и `0x2` (см. Рисунок 19).

— `0x0` в $(k+1)$-ом байте блока означает, что этот блок выделен и пока не освобождён. Кроме того, дальше в куче NVRAM также есть выделенные, но не освобождённые блоки.

— `0x1` в $(k+1)$-ом байте блока означает, что этот блок выделен и пока не освобождён. Кроме того, дальше в куче NVRAM все блоки свободны. Если $(k+1)$-ый байт N-ого блока равен `0x1`, то дальнейшие блоки ($(N+1)$-ый, $(N+2)$-ой и так далее) свободны независимо от того, что написано в их $(k+1)$-ых байтах.

— `0x2` в $(k+1)$-ом байте блока означает, что этот блок освобождён и может быть выдан пользователю повторно. Кроме того, дальше в куче NVRAM также есть выделенные, но не освобождённые блоки.

Рисунок 19 – Структура кучи NVRAM во втором алгоритме аллокации

В дальнейшем будем считать, что первый блок всегда занят (но указатель на него не выдаётся никому) и никогда не освобождается. Значит, в $(k+1)$-ом байте кучи NVRAM всегда находится либо `0x0`, либо `0x1`. Таким образом, инициализация аллокатора состоит в необходимости записать `0x1` в $(k+1)$-ый байт кучи NVRAM.

Состояние аллокатора характеризуется порядковым номером последнего выделенного блока (то есть такого блока, в $(k+1)$-ом байте которого записано `0x1`) и множеством свободных блоков, расположенных слева от адреса начала последнего выделенного блока (то есть таких блоков, расположенных до начала последнего выделенного блока, в $(k+1)$-ом байте которых записано `0x2`). И порядковый номер последнего выделенного блока (назовём это число

границей аллокации) и множество свободных блоков расположены в энергозависимой памяти.

Как и в предыдущем случае, будем защищать аллокатор от двух и более одновременных запросов на выделение или освобождение памяти с помощью грубой блокировки, расположенной в энергозависимой памяти.

3.4. Второй алгоритм: выделение памяти

Если аллокатору приходит запрос на выделение блока памяти, возможны две ситуации:

— Множество свободных блоков (то есть блоков, в $(k+1)$-ом байте которых записано `0x2`) не пустое. В таком случае, необходимо взять произвольный блок из множества свободных блоков, в $(k+1)$-ый байт этого блока записать `0x0` вместо `0x2`, удалить этот блок из множества свободных блоков и вернуть адрес его начала вызывающему.

— Множество свободных блоков пустое (см. Рисунок 20). В таком случае необходимо сдвинуть границу аллокации вправо. Пусть текущая граница аллокации равна N. Для того, чтобы сдвинуть границу аллокации вправо, сначала запишем в $(k+1)$-ый байт $(N+1)$-ого блока `0x1` (см. Рисунок 21), а потом сменим значение $(k+1)$-ого байта N-ого блока с `0x1` на `0x0` (см. Рисунок 22).

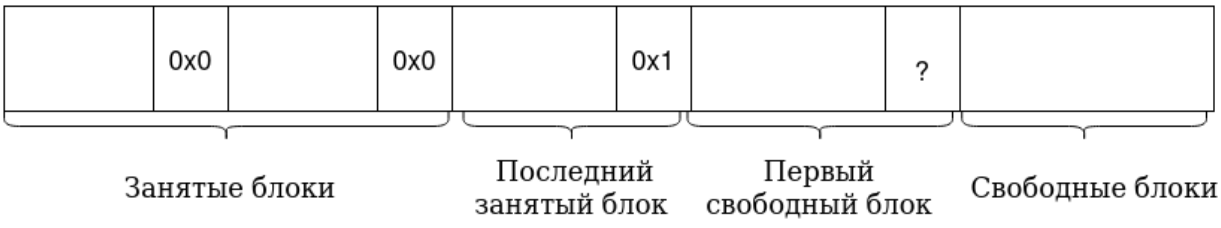

Рисунок 20 – Структура кучи NVRAM до начала перемещения границы аллокации вправо

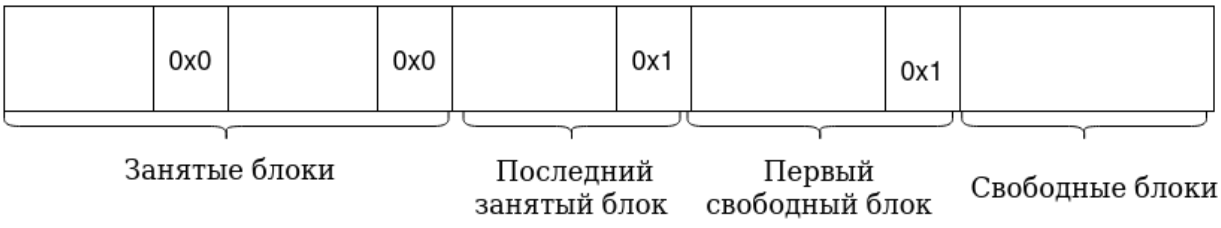

Рисунок 21 – Структура кучи NVRAM после записи `0x1` в $k+1$-ый байт первого свободного блока

Рисунок 22 – Структура кучи NVRAM после перемещения границы аллокации вправо

3.5. Второй алгоритм: освобождение памяти

Если алгоритму приходит запрос на освобождение блока памяти, возможны два варианта:

— Блок памяти, который требуется освободить, не является самым правым выделенным блоком. В таком случае в $(k+1)$-ом байте этого блока записано `0x0`. Тогда просто запишем в $(k+1)$-ый байт этого блока `0x2` и добавим этот блок в множество свободных.

— Блок памяти, который требуется освободить, является самым правым выделенным блоком (см. Рисунок 23). В таком случае необходимо переместить влево границу аллокации. Для этого необходимо найти самый правый выделеный блок, не совпадающий с блоком, который необходимо освободить (это будет самый правый блок, в $(k+1)$-ом байте которого записано `0x0`). Этот блок и станет новой границей аллокации, необходимо записать `0x1` в его $(k+1)$-ый байт. Очевидно, между новой и старой границей аллокации могут быть несколько свободных блоков (в $(k+1)$-ом байте этих блоков записано `0x2`). После перемещения границы аллокации влево эти блоки окажутся справа от границы аллокации, поэтому их необходимо удалить из множества свободных блоков (см. Рисунок 24).

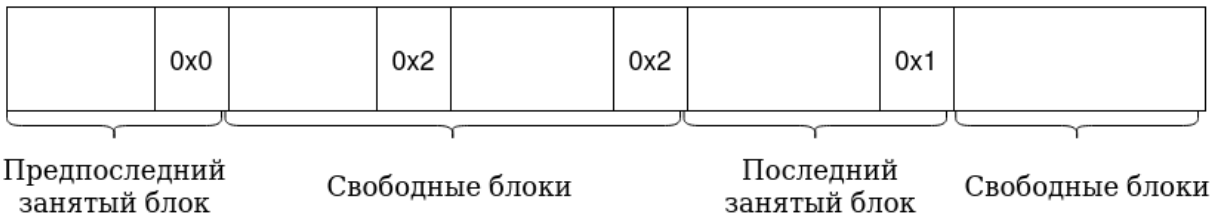

Рисунок 23 – Структура кучи NVRAM до перемещения границы аллокации влево

Рисунок 24 – Структура кучи NVRAM после перемещения границы аллокации влево

3.6. Второй алгоритм: сбои системы

Заметим, что вышеописанный алгоритм требует гарантии атомарности записи всего одного байта, поэтому его можно использовать на системах, использующих жёсткие диски в качестве энергонезависимого хранилища. Куча NVRAM никогда не будет приведена в неконсистентное состояние из-за того, что сбой системы произошёл в процессе записи.

Если произойдёт сбой системы, состояние аллокатора (граница аллокации и список свободных блоков), хранившееся в энергозависимой памяти, будет потеряно. Чтобы восстановить его, необходимо после запуска системы пройти по куче NVRAM от начала до первого блока, у которого в $(k+1)$-ом байте записано `0x1`. Порядковый номер этого блока станет границей аллокации, а порядковые номера всех встреченных в процессе обхода блоков, у которых в $(k+1)$-ом байте записано `0x2`, сформируют множество свободных блоков.

Кроме того, сбой системы может привести к утечке памяти. Представим, например, ситуацию, когда некий блок в куче NVRAM был помечен как выделенный (например, мы изменили значение его $(k+1)$-ого байта с `0x2` на `0x0`), и в этот момент произошёл сбой системы (до того, как мы успели вернуть вызывающей стороне адрес выделенного блока). Для борьбы с этим можно применить подход, описанный в работе [28]. Этот подход заключается в запуске процедуры сборки мусора после возобновления работы системы, которая позволяет разметить недоступные, но не освобождёные, блоки памяти. После разметки таких блоков эти блоки можно освободить.

3.7. Выбор используемого для реализации системы алгоритма

В работе нам необходимо эмулировать NVRAM на системах, использующих жёсткие диски в качестве энергонезависимого хранилища. Эти системы

могут не предоставлять гарантию атомарности записи в энергонезависимое хранилище восьми байт, выровненных по размеру кэш-линии, но предоставляют гарантию атомарности записи одного байта. Следовательно, при использовании первого алгоритма, куча NVRAM может прийти в неконсистентное состояние, если произойдёт сбой в процессе записи восьми байт, определяющих границу аллокации в первом алгоритме.

Кроме того, заметим, что ограничение второго алгоритма (возможность выделять только блоки памяти фиксированного размера) не является существенным ограничением при реализации системы, так как выделение динамической памяти в куче NVRAM используется только под ответ задачи `CAS` фиксированного размера (по одному байту на каждый ответ).

Следовательно, при реализации системы будет использоваться второй алгоритм выделения памяти в куче NVRAM.

Выводы по главе 3

В этой главе была рассмотрена задача выделения памяти в куче NVRAM и разработаны два алгоритма выделения памяти. Первый алгоритм допускает выделение блоков памяти произвольного размера, но не поддерживает освобождение выделенной памяти и требует для работы гарантии атомарности записи восьми байт, выровненных по размеру кэш-линии. Второй алгорим поддерживает выделение блоков памяти только фиксированного размера, зато этот алгоритм поддерживает особождение выделенной памяти и требует для корректной работы гарантию атомарности записи только одного байта.

В силу специфики работы, при реализации системы будет использоваться второй алгоритм.

ГЛАВА 4. ПРАКТИЧЕСКОЕ ПРИМЕНЕНИЕ

4.1. CAS для NVRAM

Алгоритм `CAS` для NVRAM, удовлетворяющий условию корректности Nesting-safe recoverable linearizability, описан в статье [2].

В статье предполагается отсутствие энергозависимых кэшей у NVRAM, поэтому после каждой записи в NVRAM будем немедленно производить сброс кэшей чтобы записанные данные попали в NVRAM немедленно после записи.

Для корректности описанного в статье алгоритма требуется, чтобы каждая запись в NVRAM выполнялась атомарно. Так как атомарно в NVRAM можно сбросить только одну кэш-линию, потребуем чтобы ни одна запись не пересекала границ кэш-линии.

В ходе исполнения алгоритма запись может быть произведена только в `C` — RMW-регистр, по которому делается CAS и в $R_{i,j}$ — один из single-reader-single-writer (SRSW) регистров из квадратной марицы `R`. $R_{i,j}$ используется потоком t_j для того, чтобы уведомить поток t_i о том, что осуществлённый потоком t_i `CAS` был успешен.

Пусть адрес начала `C` выровнен по размеру кэш-линии, сам RMW-регистр `C` занимает восемь байт. В таком случае, регистр `C` целиком помещается в единственной кэш-линии архитектуры x86, не пересекая её границ (так как в архитектуре `x86` кэш-линии имеют размер 64 байта). Таким образом, запись RMW-регистра `C` приводит к сбросу в NVRAM всего одной кэш-линии, и, следовательно, выполняется атомарно.

Пусть матрица `R` хранится по строкам, то есть сначала в памяти уложена первая строка матрицы, потом вторая, потом третья, и так до `N`-ой (см. Рисунок 25).

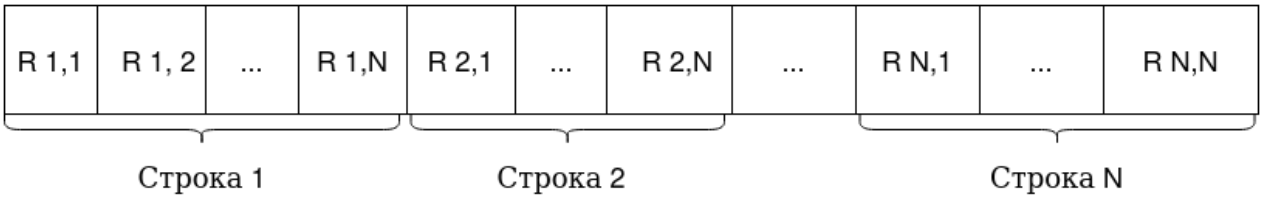

Рисунок 25 – Расположение матрицы SRSW-регистров в NVRAM

Если каждый из SRWR регистров $R_{i,j}$ занимает четыре байта, а начало матрицы выровнено по размеру кэш-линии, то ни один из регистров не пересекает кэш-линию ни при каком значении N. Таким образом, запись в любой

из SRSW-регистров $R_{i,j}$ приводит к сбросу в NVRAM всего одной кэш-линии, а значит, выполняется атомарно.

4.2. Задача тестирования CAS

Рассмотрим последовательность операций $\{CAS(var, expected_j, new_j)\}_{j=1}^{M}$, исполняемых потоками $\{t_i\}_{i=1}^{N}$. Заметим, в частности, что все операции выполняются на одном регистре.

Рассмотрим, кроме того, бинарное отношение happens before на этих операциях. Это отношение подробно описано, например, в статье [13]. Если операции a и b связаны этим отношением в исполнении H, будем обозначать это $HB_H(a, b)$ или $a \rightarrow_H b$.

Пусть мы хотим проверять исполнение операций $\{CAS(var, expected_i, new_i)\}_{j=1}^{M}$ на линеаризуемость.

Назовём исполнение H линеаризуемым, если оно эквивалентно какому-то последовательному исполнению L, при этом из $a \rightarrow_H b$, должно следовать $a \rightarrow_L b$. Более подробно данное условие корректности рассмотрено в статье [9].

Так как проверка исполнения на линеаризуемость в некоторых случаях является слишком сложной задачей, рассмотрим также сериализуемость — более слабое условие корректности в многопоточной среде. Назовём исполнение H сериализуемым, если оно эквивалентно какому-то последовательному исполнению L. Заметим, что из $a \rightarrow_H b$ может не следовать $a \rightarrow_L b$. Более подробно это условие корректности рассмотрено в статье [18].

Применим разработанный в предыдущих главах алгоритм запуска программ для NVRAM для проверки на сериализуемость и линеаризуемость алгоритма CAS, описанного в статье [2].

В дальнейшем будем называть $init$ начальное значение RMW-регистра, по которому делается CAS (то есть то значение, которое было записано в регистре до выполнения любой из операции).

4.3. Тестирование CAS с ограничениями

Для того, чтобы проверить исполнение на линеаризуемость, введём на операции несколько дополнительных ограничений.

— Каждое значение new_j из $\{\{CAS(var, expected_j, new_j)\}_{j=1}^{M}$ должно быть уникальным, то есть для любого значения a можно найти не более одного такого j, что $new_j = a$.

— Для всех $j : new_j \neq init$, то есть ни одно значение new_j из $\{\{CAS(var, expected_j, new_j)\}_{j=1}^{M}$ не должно совпадать с исходным значением переменной. Это условие, вкупе с предыдущим, утверждает, что каждое значение, включая исходное, могло оказаться в переменной не более чем единожды.
— Для всех $j : new_j \neq expected_j$.

Если последовательность операций удовлетворяет таким ограничениям, то исполнение можно проверять на линеаризуемость.

Для проверки исполнения на линеаризуемость построим ориентированный граф $G = \langle V, E \rangle$, при этом множество вершин графа имеет вид $V = \{new_j\}_{j=1}^{M} \cup \{expected_j\}_{j=1}^{M} \cup init$. Множество рёбер E строится следующим образом: между вершинами a и b есть ребро $a \to b$, если среди исполненых операций был успешный $CAS(var, a, b)$ (то есть существует j, такое что $expected_j = a$ и $new_j = b$, и при этом $CAS(var, expected_j, new_j)$ был успешен).

Построенный граф имеет следующие свойства:
— Из того, что $new_j \neq expected_j$, следует, что в графе не может быть петель.
— В вершину, соответствующую начальному значению, не может входить ни одно ребро, так как $\forall j : new_j \neq init$. Следовательно, ни для какого a в исполнении не могло найтись успешного $CAS(var, a, init)$.
— Ни в одну вершину не может войти больше одного ребра. Пусть это не так и найдутся три такие вершины (a, b, c), что $a \to b, c \to b$. Это значит, что в исполнении был успешный $CAS(var, a, b)$ и успешный $CAS(var, c, b)$. Значит, new_j не уникальны, так как как минимум в двух различных операциях $new_j = b$.

Заметим, что если в этом графе найдутся три такие различные вершины (a, b, c), что $a \to b, a \to c$ (см. Рисунок 26), то такое исполнение нелинеаризуемо. Действительно, так как значение переменной var не могло одновременно смениться с a и на b, и на c, такое могло произойти только в одном случае:
— Пусть в какой-то момент в переменной было записано значение a.
— Потом в исполнении случился успешный $CAS(var, a, b)$, в результате которого значение сменилось на b, или успешный $CAS(var, a, c)$, в ре-

зультате которого значение сменилось на c. В силу симметричности этих случаев предположим, что произошёл успешный $CAS(var, a, b)$ и значение сменилось с a на b, а в графе появилось ребро $a \to b$.

— Потом значение снова сменилось на a.

— Потом произошёл успешный $CAS(var, a, c)$, в результате которого значение сменилось с a на c, а в графе появилось ребро $a \to c$.

Заметим, что в таком случае значение a оказалось записано в переменную более чем единожды, а по условиям такого быть не может.

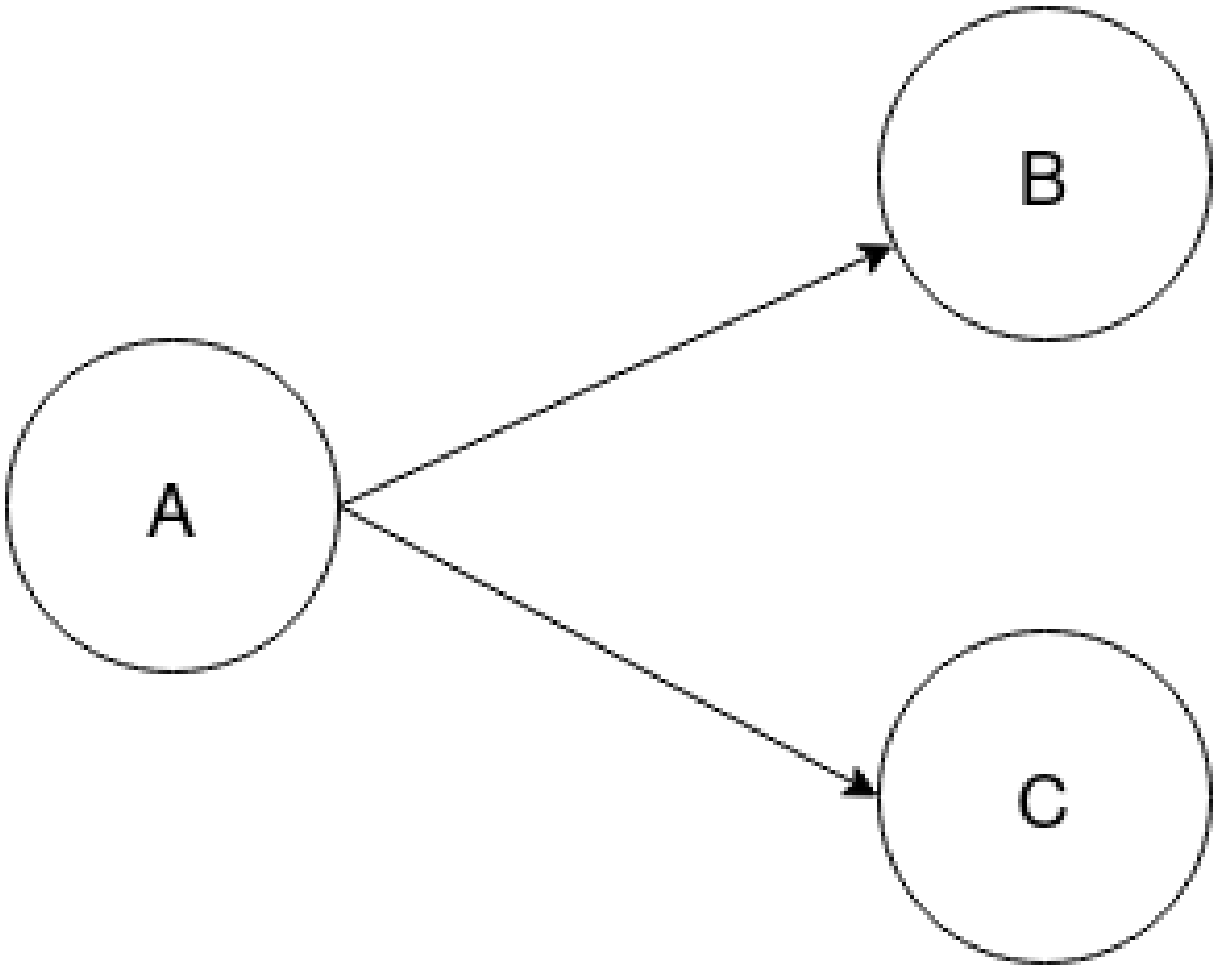

Рисунок 26 – Пример графа, соответствующего не линеаризуемому исполнению

Теперь заметим, что если в графе нашлись такие вершины a, b что $a \to b$, при этом $a \neq init$ и не существует c с $c \to a$ (см. Рисунок 27), то такое исполнение не линеаризуемо.

Действительно, исходное значение переменной не было равно a и в исполнении не было успешной операции $CAS(var, c, a)$. Таким образом, значе-

ние переменной ни в какой момент не могло стать равным a, значит, ни для какого значения b операция $CAS(var, a, b)$ не могла закончиться успешно, таким образом, в графе не может существовать ребра $a \to b$.

Рисунок 27 – Пример графа, соответствующего нелинеаризуемому исполнению

Из этого следует, что линеаризуемому исполнению соответствует только один вид графов (см. Рисунок 28): те, в которых рёбра образуют единственный эйлеров путь, начинающийся в вершине, соответствующей начальному значению и заканчивающийся в вершине, соответствующей значению регистра var после исполнения всех операций (то есть граф является ‘‘бамбуком’’). Если такой путь найти не удаётся, то исполнение точно не линеаризуемо.

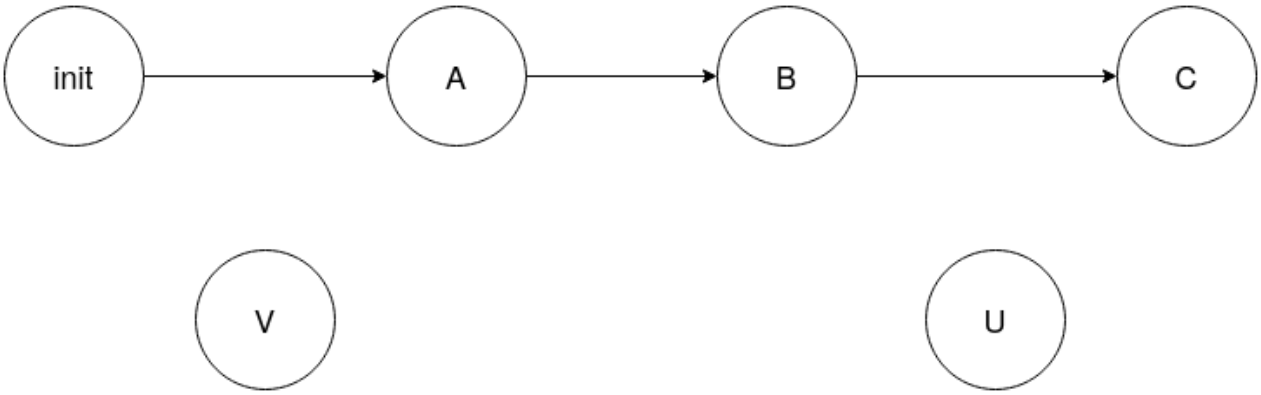

Рисунок 28 – Пример графа, возможно соответствующего линеаризуемому исполнению

Если же удалось найти такой путь, то нам удалось найти единственное возможное последовательное исполнение L, эквивалентное исполнению H. При этом не может существовать другое последовательное исполнение L', эквивалентное данному, так как в графе существует единственный эйлеров путь.

Для завершения проверки линеаризуемости остаётся только проверить, исполнение L учитывает ограничения, заданные отношением happens-before (то есть $a \to_H b \Rightarrow a \to_L b$).

Заметим, что исполнение L последовательное, поэтому оно задаёт полный порядок на операциях. Поэтому для того, чтобы проверить, что отноше-

ние L удовлетворяет ограничениям happens-before нужно для каждой пары операций

$$CAS(var, a, b), CAS(var, c, d) : \begin{cases} CAS(var, a, b) \rightarrow_H CAS(var, c, d) \\ CAS(var, a, b) = true \\ CAS(var, c, d) = true \end{cases}$$

проверить, что ребро $a \rightarrow b$ в найденном эйлеровом пути идёт до ребра $c \rightarrow d$. Еси это верно, то исполнение H линеаризуемо. Если же это неверно, то исполнение H не линеаризуемо, но сериализуемо (так как существует последовательное исполнение L, эквивалентное исполнению H, но не существует последовательного исполнения L', эквивалентного исполнению H и учитывающего ограничения, заданные отношением happens-before).

4.4. Тестирование CAS

Можно отказаться от ограничений, введённых в начале предыдущего пункта, и проверять на сериализуемость исполнение произвольной последовательности $\{CAS(var, expected_j, new_j)\}_{j=1}^{M}$.

Построим ориентированный граф $G = \langle V, E \rangle$ по тем же принципам, что и в предыдущем пункте. В этот раз из-за отсутствия наложенных в предыдущем пункте ограничений в графе могут быть:

— Петли, так как разрешены операции вида $CAS(var, a, a)$
— Такие тройки различных вершин (a, b, c), что $a \rightarrow b, a \rightarrow c$, так как значение могло быть записано в переменную больше одного раза.
— Такие тройки различных вершин (a, b, c), что $a \rightarrow b, c \rightarrow b$, так как нет ограничения на уникальность new_j и можно найти более одного такого j, что $new_j = b$.
— Более одного ребра $a \rightarrow b$, так как в исполнении могло быть более одной успешной операции $CAS(var, a, b)$.

Теперь для того, чтобы проверить сериализуемость исполнения, необходимо найти произвольное последовательное исполнение, эквивалентное полученному, при этом найденное последовательное исполнение может не учитывать ограничения, заданные отношением happens-before.

Рассмотрим произвольный путь, проходящий по графу G. Каждое ребро $a \rightarrow b \in E$ сответствует исполнению одной операции $CAS(var, a, b)$. Так

как каждый успешный CAS был исполнен ровно один раз, нужно найти такой путь в графе, который проходит по каждому ребру ровно один раз, то есть найти эйлеров путь. При этом известно, что начальное значение регистра var было равно $init$, а значит найденный эйлеров путь должен начинаться в вершине, соответствующей значению $init$. При этом нам так же известно значение $finish$ — значение регистра var после исполнения всех операций. Значит, найденный эйлеров путь должен заканчиваться в вершине, соответствующей значению $finish$.

В таком случае проверка исполнения на сериализуемость сводится к нахождению эйлерова пути, начинающегося в вершине, соответствующей значению $init$ и заканчивающегося в вершине, соответствующей значению $finish$.

Для проверки на сериализуемость используем следующий алгоритм:

— Если более одной компоненты слабой связности содержат хотя бы одно ребро, то исполнение несериализуемо, так как в таком случае в графе не существует ни одного эйлерова пути.

— Для каждой вершины v считаем количество входящих в неё и исходящих из неё рёбер $deg^+(v), deg^-(v)$.

— Если $deg^+(v) = deg^-(v)$ для любого $v \in V$, в графе существует эйлеров цикл. Если вершина, соответствующая значению $init$, входит в этот цикл и при этом $init = finish$, значит, исполнение сериализуемо — выберем эйлеров путь таким образом, чтобы он начинался и заканчивался в вершине, соответствующей значению $init$. В противном случае исполнение не сериализуемо.

— Если для любого $v \in V \setminus \{a, b\} : deg^+(v) = deg^-(v)$, $deg^+(a) = deg^-(a) - 1, deg^+(b) = deg^-(b) + 1$, то исполнение сериализуемо только в том случае, если вершина a соответствует значению $init$, а вершина b соответствует значению $finish$ (так как найденный эйлеров путь может начинаться только в вершине a, а заканчиваться только в вершине b).

4.5. Характеристики и результаты тестирования

Тестирование алгоритма `CAS` для NVRAM происходило в следующем окружении:

— Для тестирования использовалось четыре рабочих потока.

— NVRAM эмулировалась с помощью файлов, отображаемых в память. В качестве энергонезависимого хранилища использовались жёсткие диски. Для сброса кэша из оперативной памяти в энергонезависимое хранилище использовался системный вызов `msync` [17].
— Сбой системы эмулировался с помощью посылания сигнала `SIGKILL` всему процессу (то есть прерывался главный поток и все рабочие потоки). Для посылки процессу сигнала `SIGKILL` использовалась утилита `kill` [12].
— Для того, чтобы операцию `CAS` можно было успеть прервать в процессе её выполнения (и таким образом проверить, что даже сбой системы в процессе выполнения операции не нарушает линеаризуемости исполнения), в код операции `CAS` было помещено несколько инструкций `sleep`, позволяющих замедлить выполнение операции.

Алгоритм `CAS` из статьи [2] был проверен на нескольких десятках сценариев, при этом учитывались ограничения из секции 4.3. Каждое из полученных в ходе тестирования исполнений оказалось линеаризуемо.

4.6. Тестирование некорректного алгоритма CAS

Рассмотрим некорректный алгоритм `CAS` для NVRAM, в котором отсутствует матрица `R`, обеспечивающая связь между читателями и писателями. Считаем, что `Var` - 32х-битный регистр, расположенный в NVRAM. Псевдокод этого алгоритма может выглядеть следующим образом (см. Листинг 2):

Листинг 2 – Псевдокод некорректного алгоритма CAS

```
function cas_incorrect(Var, expected, new)
    result = CAS(Var, expected, new)
    if result then
        pmem_flush(Var, 4)
        return true
    else
        return false
    end if
end function

function cas_incorrect_recover(Var, expected, new)
    uint32_t cur_value
    pmem_read(Var, &cur_value, 4)
    if cur_value == new then
        return true
    else
        return cas_inocrrect(Var, expected, new)
    end if
end function
```

Рассмотрим следующее исполнение, нескольких операций `CAS` двумя потоками (`A` и `B`).

— Пусть изначальное значение переменной равно 42
— Поток `A` исполняет `CAS(Var, 42, 24)`.
— Одновременно поток `B` исполняет `CAS(Var, 24, 14)`.
— Поток `A` успешно завершает свою операцию (значение переменной меняется с 42 на 24), но засыпает, не вернув `true` вызывающей стороне.
— Поток `B` успешно завершает свою операцию (значение переменной меняется с 24 на 14) и возвращает значение. `true` вызывающей стороне
— В этот момент происходит сбой системы
— После сбоя системы в потоке `A` вызывается операция `cas_incorrect_recover(Var, 42, 24)`, возвращающая `false` (так как текущее значение переменной `Var` равно 14).

Таким образом, соответствующий исполнению граф выглядит следующим образом (см. Рисунок 29):

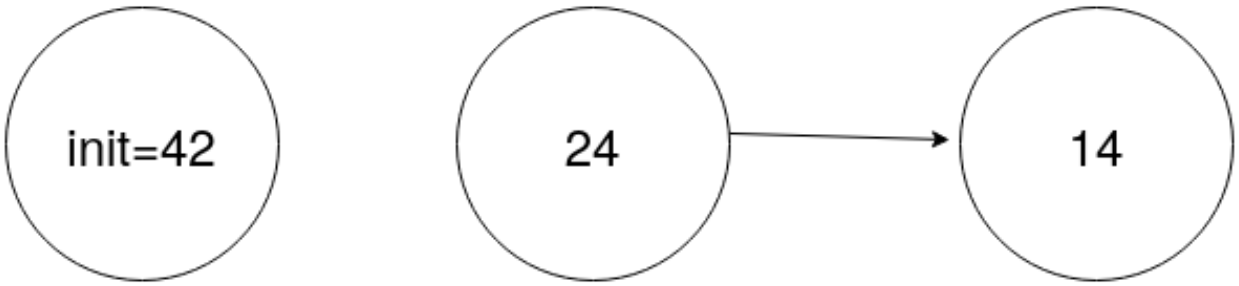

Рисунок 29 – Граф, соответствующий исполнению

Как показано в секции 4.3, это исполнение не линеаризуемо. Потребовалось три запуска разработанной мной системы, чтобы обнаружить нелинеаризуемое исполнение некорректного алгоритма `CAS` для NVRAM.

Выводы по главе 4

В этой главе было рассмотрено применение разработанного алгоритма для тестирования алгоритма `CAS` для NVRAM. Рассмотрены два метода проверки алгоритма `CAS` для NVRAM на корректность: с помощью первого метода можно проверить исполнение `CAS` с ограничениями на линеаризуемость, а с помощью второго метода можно проверить исполнение `CAS` без ограничений на сериализуемость. Кроме того, в этой главе был описан процесс тестирования алгоритма `CAS` для NVRAM, описанного в статье [2], и тестрования некорректного алгоритма `CAS` для NVRAM.

ЗАКЛЮЧЕНИЕ

Мне удалось разработать алгоритм запуска программ для NVRAM, который запускает программы в модели Nesting-safe recoverable linearizability и восстанавливает систему после сбоя. Этот алгоритм удалось проэмулировать без доступа к NVRAM, с использованием только жёстких дисков. Следовательно, разработанный алгоритм можно использовать как для запуска программ в системе с использованием NVRAM, так и в системе с использованием жёстких дисков в качестве энергонезависимого хранилища. Например, разработаный алгоритм можно использовать для тестирования алгоритмов для NVRAM без доступа к железу с NVRAM.

Кроме того, мне удалось протестировать алгоритм `CAS` для NVRAM на корректность, используя разработанную мной систему.

Можно предложить несколько путей развития и улучшения существующего решения. Например, можно выполнять следующие преобразования пользовательского кода:

— Автоматически выполнять сериализацию/десериализацию параметров функции в массив байт и из массива байт, позволяя пользователю принимать и передавать аргументы привычным способом.
— Автоматически регистрировать все необходимые функции в глобальной хеш-таблице.
— Позволить пользователю вызывать функцию привычным образом, а не через `do_call`, автоматически вставляя код для добавления нового фрейма и перемещения конца стека.
— Автоматически вставлять код для сохранения возвращаемого значения.
— Автоматически вставлять код для преобразования указателей на NVRAM в смещения от начала отображения NVRAM в виртуальное адресное пространство.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 A persistent lock-free queue for non-volatile memory / M. Friedman [et al.] // ACM SIGPLAN Notices. — 2018. — Vol. 53, no. 1. — P. 28–40.

2 *Attiya H.*, *Ben-Baruch O.*, *Hendler D.* Nesting-safe recoverable linearizability: Modular constructions for non-volatile memory // Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing. — 2018. — P. 7–16.

3 compare-and-swap [Электронный ресурс]. — 2020. — URL: `https://en.wikipedia.org/wiki/Compare-and-swap`.

4 *Denning P. J.* Virtual memory // ACM Computing Surveys (CSUR). — 1970. — Vol. 2, no. 3. — P. 153–189.

5 fetch-and-add [Электронный ресурс]. — 2020. — URL: `https://en.wikipedia.org/wiki/Fetch-and-add`.

6 File locking [Электронный ресурс]. — 2020. — URL: `https://en.wikipedia.org/wiki/File_locking`.

7 gtest [Электронный ресурс]. — 2020. — URL: `https://github.com/google/googletest`.

8 *Herlihy M.* Wait-free synchronization // ACM Transactions on Programming Languages and Systems (TOPLAS). — 1991. — Vol. 13, no. 1. — P. 124–149.

9 *Herlihy M. P.*, *Wing J. M.* Linearizability: A correctness condition for concurrent objects // ACM Transactions on Programming Languages and Systems (TOPLAS). — 1990. — Vol. 12, no. 3. — P. 463–492.

10 *Izraelevitz J.*, *Mendes H.*, *Scott M. L.* Linearizability of persistent memory objects under a full-system-crash failure model // International Symposium on Distributed Computing. — Springer. 2016. — P. 313–327.

11 *Kerrisk M.* The Linux Programming interface. — San Francisco : William Pollock, 2010. — 1556 p.

12 kill(1) [Электронный ресурс]. — 2020. — URL: `https://man7.org/linux/man-pages/man1/kill.1.html`.

13 *Lamport L.* Time, clocks, and the ordering of events in a distributed system // Concurrency: the Works of Leslie Lamport. — 2019. — P. 179–196.

14 malloc(3) [Электронный ресурс]. — 2020. — URL: https://man7.org/linux/man-pages/man3/malloc.3.html.

15 mmap(2) [Электронный ресурс]. — 2020. — URL: http://man7.org/linux/man-pages/man2/mmap.2.html.

16 Monitor [Электронный ресурс]. — 2020. — URL: https://en.wikipedia.org/wiki/Monitor_(synchronization).

17 msync(2) [Электронный ресурс]. — 2020. — URL: http://man7.org/linux/man-pages/man2/msync.2.html.

18 *Papadimitriou C. H.* The serializability of concurrent database updates // Journal of the ACM (JACM). — 1979. — Vol. 26, no. 4. — P. 631–653.

19 pipe(2) [Электронный ресурс]. — 2020. — URL: https://man7.org/linux/man-pages/man2/pipe.2.html.

20 PMDK [Электронный ресурс]. — 2020. — URL: https://github.com/pmem/pmdk/.

21 POSIX message queues [Электронный ресурс]. — 2020. — URL: https://www.man7.org/linux/man-pages/man7/mq_overview.7.html.

22 POSIX semaphores [Электронный ресурс]. — 2020. — URL: https://man7.org/linux/man-pages/man7/sem_overview.7.html.

23 Process [Электронный ресурс]. — 2020. — URL: https://en.wikipedia.org/wiki/Process_(computing).

24 Program counter [Электронный ресурс]. — 2020. — URL: https://en.wikipedia.org/wiki/Program_counter.

25 test-and-set [Электронный ресурс]. — 2020. — URL: https://en.wikipedia.org/wiki/Test-and-set.

26 Thread [Электронный ресурс]. — 2020. — URL: https://en.wikipedia.org/wiki/Thread_(computing).

27 Thread-local storage [Электронный ресурс]. — 2020. — URL: https://en.wikipedia.org/wiki/Thread-local_storage.

28 Understanding and optimizing persistent memory allocation / W. Cai [et al.] // Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. — 2020. — P. 421–422.

29 x86 calling conventions [Электронный ресурс]. — 2020. — URL: https://en.wikipedia.org/wiki/X86_calling_conventions.