

Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 21.Б07-мм

Оптимизация порта macOS/AArch64 в OpenJDK

Целовальникова Екатерина Андреевна

Отчёт по учебной практике
в форме «Решение»

Научный руководитель:
ассистент кафедры СП А. П. Козлов

Консультант:
ассистент кафедры СП А. П. Козлов

Санкт-Петербург
2023

Оглавление

Введение	3
1. Постановка задачи	4
2. Обзор	5
2.1. Hardened Runtime [5]	5
2.2. Портирование JIT-компиляторов на Apple Silicon [7]	5
2.3. Реализация W^X в JEP 391	7
3. Ход работы	9
3.1. Процесс разработки OpenJDK	9
3.2. Реализация ленивого способа переключения состояния W^X	10
4. Тестирование	14
Заключение	15
Список литературы	16

Введение

Java является одним из самых известных языков программирования в мире на протяжении многих лет. В настоящее время насчитываются миллионы профессиональных разработчиков, которые используют её в качестве основного языка. Java наиболее востребована и полезна в таких направлениях, как веб-разработка, компьютерные программы, приложения для мобильных устройств и промышленное программирование.

Для создания и тестирования программ, написанных на языке Java, необходима разработка спецификации платформы, называемая JDK (Java Development Kit). Эталонной реализацией Java является проект по созданию полностью совместимого JDK – OpenJDK. Он состоит исключительно из свободного и открытого исходного кода.

В июне 2020 года компания Apple объявила о своих планах по процессу внедрения Apple Silicon на основе архитектуры AArch64 в линейке компьютеров Apple Macintosh. Для предоставления полноценной и сертифицированной версии JDK на платформах AArch64 в macOS был разработан JEP (Java Enhancement Proposal) 391. Благодаря портированию на данную архитектуру список дистрибутивов OpenJDK пополнился сборками для чипа Apple M1.

Порт Apple M1 использует политику защиты памяти W^X (Write xor Execute), при которой каждая страница в адресном пространстве процесса или ядра может быть доступна только для записи или только для исполнения.

В качестве оптимизации порта macOS/AArch64 рассматривается возможность переключения состояния функции безопасности W^X только в случае неверного обращения к памяти процессора.

1. Постановка задачи

Целью данной работы является исследование альтернативной реализации управлением состояния функции безопасности W^X . Для достижения цели были поставлены следующие задачи:

1. Сделать обзор функции Hardened Runtime и портирования JIT-компиляторов на Apple Silicon
2. Сделать обзор управления памяти W^X в реализации JEP 391
3. Ознакомиться с процессом разработки OpenJDK
 - 3.1 Сборка OpenJDK на x86-64 на локальной машине
 - 3.2 Сборка OpenJDK на Apple M1 на удаленной машине
 - 3.3 Рассмотреть инструменты изменения и отладки кода OpenJDK
4. Реализовать ленивый способ переключения состояния W^X
 - 4.1 Исследовать влияние переключения состояния W^X
 - 4.2 Исследовать предыдущую реализацию переключения состояния W^X
 - 4.3 Реализовать альтернативное переключение состояния W^X
5. Оценить производительность до и после внесенных изменений

2. Обзор

2.1. Hardened Runtime [5]

Функция Hardened Runtime осуществляет управление защитой безопасности и доступом к ресурсам приложений macOS. Hardened Runtime вместе с System Integrity Protection (SIP) [8] защищает целостность программного обеспечения во время выполнения, предотвращая определенные классы эксплойтов, такие как внедрение кода, захват динамически связанных библиотек и подделка пространства памяти процессов.

Hardened Runtime не оказывает влияние на работу большинства приложений, но она запрещает некоторые менее распространенные возможности, такие как Just-In-Time (JIT) компиляция. В случае, если приложение полагается на возможность, которую ограничивает Hardened Runtime, необходимо добавить право на отключения индивидуальной защиты.

2.2. Портирование JIT-компиляторов на Apple Silicon [7]

2.2.1. Описание JIT компиляции

Компилятор Just-In-Time (JIT) переводит байт-код или промежуточный код скрипта в инструкции машинного языка и делает эти инструкции доступными для выполнения. Приложение начинает компиляцию JIT по мере необходимости для поддержки соответствующих задач. Процесс компиляции происходит в пространстве процесса приложения.

Поскольку компиляция JIT использует методы, которые запрещаются функцией Hardened Runtime, перед портированием необходимо обновить приложение, если оно поддерживает эту возможность.

2.2.2. Включение прав JIT для Apple Silicon

Функция Hardened Runtime запрещает выполнение кода на странице памяти, если этот код не сопровождается действительной подписью. Поскольку JIT-компилятор не подписывает генерируемый им код, он технически нарушает правила Hardened Runtime. Для устранения этой проблемы при портировании приложения необходимо разрешить в настройках Hardened Runtime действия, связанные с JIT компиляцией.

Это необходимо для использования одного и того же участка памяти для записи или для исполнения. Это участок памяти должен быть создан с помощью вызова метода `mmap()`, с использованием параметра `MAP_JIT` [1]. В результате, во время выполнения поток сможет производить доступ для записи или исполнения в зависимости от последнего вызова функции `pthread_jit_write_protect_np()`.

2.2.3. Отключение защиты записи перед выполнением инструкций

Когда защита памяти включена, поток не может записывать в область памяти и выполнять инструкции в этой области одновременно. Apple Silicon обеспечивает защиту памяти для всех приложений, независимо от того, используют ли они Hardened Runtime. Стоит отметить, что компьютеры Mac на базе Intel обеспечивают защиту памяти только для приложений, использующих Hardened Runtime [7].

2.2.4. Восстановление защиты от записи перед выполнением инструкций

Необходимо, чтобы JIT компилятор отключил защиту перед записью машинных инструкций в область памяти. При выполнении любой из записанных им инструкций необходимо предварительно восстановить используемые средства защиты.

2.3. Реализация W^X в JEP 391

Класс ThreadWXEnabel [3], осуществляющий исполнение политики безопасности локальных потоков Java, является реализацией программной идиомы RAII. Пример его использования в коде программного проекта OpenJDK представлен в файле safepoint.cpp [4]. Функция класса ThreadWXEnabel вызывается обработчиком сигнала в произвольной точке выполнения кода и отвечает за переключение состояния W^X .

Листинг 1: Переменные класса ThreadWXEnabel

```
Thread* _thread;  
WXMode _old_mode;
```

Переменная `_thread` является потоком, который отвечает за исполнение инструкций запущенного процесса. А переменная `_old_mode` несет ответственность за состояния функции безопасности W^X , которое было определено при создании или деинициализации объекта.

Ниже представлена реализация конструктора и деструктора ThreadWXEnabel. По их коду становится понятно, что при создании и деинициализации объекта рассматриваемого класса происходит смена состояния функции безопасности W^X .

Листинг 2: Конструктор ThreadWXEnabel

```
ThreadWXEnable(WXMode new_mode, Thread* thread) :  
    _thread(thread),  
    _old_mode(_thread ? __thread → enable_wx(new_mode) :  
    WXWrite) { }
```

Листинг 3: Деструктор ThreadWXEnabel

```
~ ThreadWXEnable( )  
{  
    if (_thread)  
        _thread → enable_wx(_old_mode);  
};
```

Рассматривая интерфейс `enable_wx` можно заметить, что осуществляет вызов `os::current_thread_enable_wx(new_state)`. Именно эта строка отвечает за получение доступа потока выполнения для записи или исполнения.

Листинг 4: Интерфейс `enable_wx`

```
inline WXMode Thread::enable_wx(WXMode new_state)
{
    assert(this == Thread::current(),
           "should only tbe called for current thread");
    assert(_wx_init, "should be inited");
    WXMode old = _wx_state;
    if (_wx_state != new_state)
    {
        _wx_state = new_state;
        os::current_thread_enable_wx(new_state);
    }
    return old;
}
```

Листинг 5: Реализация `enable_wx`

```
void os::current_thread_enable_wx(WXMode mode)
{
    pthread_jit_write_protect_np(mode == WXExec);
}
```

3. Ход работы

3.1. Процесс разработки OpenJDK

При выполнении работы над проектом была поставлена задача собрать JDK самостоятельно для подробного ознакомления с требованиями и функциями сборки OpenJDK и дальнейшей работы с программным проектом.

3.1.1. Сборка OpenJDK на x86-64

В качестве тренировочного упражнения сборка JDK первоначально осуществлялась на процессоре Apple x86-64 на локальной машине. Шаги, последовательно выполненные в процессе сборки OpenJDK, подробно описаны на сайте OpenJDK Git [6].

Перед началом сборки потребовалось скачать на локальную машину утилиту командной строки homebrew. Homebrew является бесплатной системой управления пакетами программного обеспечения с открытым исходным кодом, которая упрощает установку программного обеспечения на операционную систему macOS.

Следует отметить, что для создания JDK потребовался уже существующий JDK. Другими словами, Java использовала метод создания транслятора для языка программирования Java, который называется раскруткой компилятора.

Важным моментом при сборке являлось существование двух целей компиляции: `make images` и `make`, аналогичный `make default` или `make jdk`. Различие этих команд состоит в том, что один из них является сборкой полного дистрибутива Java, а другой запускает только что собранную Java без использования самой себя. Таким образом, использование команды `make` оказалось очень полезным в случае возникновения проблем, препятствующих работе Java.

3.1.2. Сборка OpenJDK на процессоре Apple M1

Для разработки на машине с процессором Apple M1 потребовалось подключиться к серверу, используя пару SSH-ключей. На удаленной машине следовало создать рабочую директорию, в которой выполнялись дальнейшие действия по оптимизации порта Apple M1.

Следует подчеркнуть, что после подключения к серверу потребовалось сделать инициализацию системы управления пакетами программного обеспечения homebrew. Для этого была использована команда `eval "$ (homebrew/bin/brew shellenv)"`.

Для реализации процесса сборки OpenJDK на удаленной машине выполнены шаги, идентичные тем, что были использованы для сборки OpenJDK на локальной машине [6].

3.1.3. Инструменты, использованные для изменения и отладки кода OpenJDK

Для изменения кода порта Apple M1 использован консольный текстовый редактор Nano, который включает в себя основные команды для редактирования текста.

Для отладки кода порта Apple M1 использован отладчик LLDB, который использует анализатор выражений Clang и дизассемблер LLVM.

3.2. Реализация ленивого способа переключения состояния W^X

3.2.1. Влияние переключения состояния W^X

Для наилучшего понимания влияния переключения состояния функции безопасности W^X был проведен небольшой эксперимент. Формулировка задачи заключалась в том, чтобы закомментировать строку `os::current_thread_enable_wx` в функции `Thread::enable_wx` исполняемого файла `thread.inline.hpp` [2]. После выполнения рассматриваемого эксперимента можно убедиться, что при отключении обработки состояния W^X Java выходит из строя и не реагирует на запросы. Таким обра-

зом, было принято решение оставить строку `os::current_thread_enable_wx` закоментированной для дальнейших изменений внутренней структуры программы.

3.2.2. Исследование предыдущей реализации переключения состояния W^X

Листинг 6: Функция PosixSignals

```
bool PosixSignals::pd_hotspot_signal_handler
(int sig, siginfo_t* info, ucontext_t* uc, JavaThread* thread)
{
    ThreadWXEnable wx(WXWrite, thread);
    ...
}
```

Последующие исследования проводились в исполняемом файле `aarch64.cpp` в функции `PosixSignals`. Рассматриваемая функция осуществляет асинхронное уведомление процесса о каком-либо событии и является один из основных способов взаимодействия между процессами. Когда сигнал послан процессу, операционная система прерывает выполнение процесса, при этом, если процесс установил собственный обработчик сигнала, операционная система запускает этот обработчик, передав ему информацию о сигнале, если процесс не установил обработчик, то выполняется обработчик по умолчанию.

В прежней версии вышеуказанного файла в начале `PosixSignals` осуществлялся вызов функции `ThreadWXEnable`. Рассматриваемая функция отвечает за переключение состояния W^X на режим «записи», в независимости от того, что было установлено ранее, а на выходе из функции `PosixSignals` возвращает старый режим.

3.2.3. Реализация альтернативного переключения состояния W^X

Изменения внутренней структуры программы проводились в функции `PosixSignals` исполняемого файла `aarch64.cpp`.

Во-первых, было принято решение закомментировать вызов функции ThreadWXEnable. Из-за совершенных ранее действий данная функция больше не выполняет переключения состояния W^X , но ее вызов можно использовать для получения старого состояния функции безопасности W^X .

Во-вторых, в функции PosixSignals при возникновении сигнала необходимо было проверить текущий режим исполнения Java Virtual Machine и причину возникновения этого сигнала. Если же адрес возникшего сигнала принадлежал CodeCache и тип его обращения («запись» или «исполнение») был действителен, но не совпадал с текущим режимом W^X , следовало выполнить переключение.

В-третьих, при каждом выходе из функции PosixSignals необходимо было вернуть старый режим функции безопасности W^X , иначе сборка Java впадала в бесконечный цикл.

Листинг 7: Вызов в начале PosixSignals

```
#if defined(APPLE) && defined(AARCH64)
    void* cause = info → si_addr;
    auto old_mode = thread → enable_wx(WXWrite);
    bool switched = false;
    if (old_mode != WXWrite && CodeCache::contains(cause))
    {
        os::current_thread_enable_wx(WXWrite);
        switched = true;
    }
#endif
```

Листинг 8: Вызов перед каждым выходом из PosixSignals

```
#if defined(APPLE) && defined(AARCH64)
    if (switched)
    {
        thread → enable_wx(old_mode);
        os::current_thread_enable_wx(old_mode);
    }
#endif
```

4. Тестирование

Тестирование производительности порта Apple M1 до и после выполнения оптимизации будет выполнено с использованием эталонных тестов, которые фиксируют репрезентативные модели поведения приложений. Renaissance Benchmark Suite предоставляет такой тестовый набор, состоящий из современных, объектно-ориентированных рабочих нагрузок, которые используют различные примитивы параллелизма Java Virtual Machine (JVM).

Заключение

В ходе выполнения данной работы были выполнены следующие задачи:

1. Проведен обзор функции Hardened Runtime и портирования JIT-компиляторов на Apple Silicon
2. Проведен обзор управления памяти W^X в реализации JEP 391
3. Выполнена сборка OpenJDK на x86-64 на локальной машине
4. Выполнена сборка OpenJDK на Apple M1 на удаленной машине
5. Выполнен обзор инструментов, используемых для изменения и отладки кода OpenJDK
6. Исследовано влияние переключения состояния W^X
7. Исследована предыдущая реализация переключения состояния W^X
8. Реализовано альтернативное переключение состояния W^X

В рамках дальнейшей деятельности планируется сравнить результаты тестирования производительности до и после внесенных изменений.

Список литературы

- [1] Allow Execution of JIT-compiled Code Entitlement. — URL: https://developer.apple.com/documentation/bundleresources/entitlements/com_apple_security_cs_allow-jit?changes=__11 (дата обращения: 15.12.2022).
- [2] Calling `enable_wx`. — URL: <https://github.com/openjdk/jdk/blob/270cf67e5ff19b082f710d52831f436dd144d883/src/hotspot/share/runtime/thread.inline.hpp#L88> (дата обращения: 15.12.2022).
- [3] Class `ThreadWXEnabel`. — URL: <https://github.com/openjdk/jdk/blob/master/src/hotspot/share/runtime/threadWXSetters.inline.hpp#L35> (дата обращения: 15.12.2022).
- [4] Example of `ThreadWXEnable` call. — URL: <https://github.com/openjdk/jdk/blob/1e1db5debd5e37650d7d7345544104a9050f418c/src/hotspot/share/runtime/safepoint.cpp#L746> (дата обращения: 15.12.2022).
- [5] Hardened Runtime. — URL: https://developer.apple.com/documentation/security/hardened_runtime?changes=__11 (дата обращения: 15.12.2022).
- [6] OpenJDK Build. — URL: <https://github.com/openjdk/jdk/blob/master/doc/building.md?ysclid=lbpcfp9ts2676952319> (дата обращения: 15.12.2022).
- [7] Porting Just-In-Time Compilers to Apple Silicon. — URL: https://developer.apple.com/documentation/apple-silicon/porting-just-in-time-compilers-to-apple-silicon?changes=__11 (дата обращения: 15.12.2022).
- [8] System Integrity Protection. — URL: <https://support.apple.com/en-us/HT204899> (дата обращения: 15.12.2022).