**1. Виртуальная машина и MRE. Достоинства и недостатки MRE по сравнению с естественным кодом платформы**

Изначальное определение «виртуальная машина — изолированный дубликат или копия реальной машины» Попек и Голдберг

*Опр*: Виртуальная машина(VM) — общее программная реализация определенной машины(например, отдельный компьютер или ), исполняющее программы как данная машина.

Программы могут быть большими как ОС, или маленькими, как процесс.

Код программы может быть интерпретирован, скомпилирован (JIT — выполнение компиляции по запросу, AOT - компиляция выполнена заранее) или просто запущен под контролем VM

*Примеры*: Xen, VMWare, JVM, LLVM, Parrot, ScrummVM, DosBox, Sega Emulator

Изначально JVM задумывалась аппаратной

*Опр*: Runtime Environment(Окружение времени исполнения): Библиотека, реализующая поведение программы конкретного языка

* Вход/выход из программы
* Доступ к сервисам ОС — ввод, вывод итд
* Проверка типов во время исполнения
* Обработка исключений
* Генерация и исполнение кода

Главная функциональность, которую предоставляет ОС — многозадачность(запуск и отслеживание работы программы)

Если программа одна и без ошибок — ОС не нужна (Системы без ОС называют baremetal, «на голом железе»)

Программы на C и ASM могут запускаться без runtime, но это непрактично.

*Опр*: Managed Runtime Environment(MRE) — среда исполнения, обеспечивающая восстановление от ошибок в ходе исполнения кода

* Ошибочный код не может свободно исполняться
* Исключения должны быть брошены немедленно в контексте утверждения ЯВУ
* Безопасность типов, Безопасность указателей, Безопасность нитей, Сборка мусора, Унификация интерфейсов, Основные управляемые библиотеки, Изоляция «железа», Верификация кода, Кросс-платформенность кода

Альернативой MRE является Бинарная компиляция — трансляция кода в бинарные инструкции для целевой архитектуры

* Статическая компиляция — не надо запускать копилятор, но больший размер бинарника
* Бинарники для конкретных платформ:
  + Отимизация под конкретную платформу
  + Нет кросс-платформенных бинарников
* Уязвимости в безопасности:
  + Код семантически не проверяется на правильность
  + Уязвимость по отношению к исполняемым данным

|  |  |
| --- | --- |
| Плюсы MRE | Минусы MRE |
| * Код меньше * Кросс-платформенные бинарники * Профилирование(сбор профиля по статистике программы) * Безопаснее * Лучше дебажится * Потенциальная замена ОС | * Большая среда исполнения(300 Кб — минимум для Java) * Больше время исполнения * Более медленный запуск при JIT — компиляции(примерно в 2 раза, AOT — примерно одинаково) * Ограничена спецификация по платформам * Ограничено использование системного программирования |

Примеры MRE/VM

Специализированные

RIP for PostScript, SQL Server

Общие

PHP, Python, Ruby, CLISP, TCL, Lua, Java RE, CLI(.NET), Google V8 (Java Script), Mozilla Spider Monkey (Java Script), Android Dalvik (Java)

**2. Основные компоненты MRE и их назначение**

Загрузчик IR

Верификатор

Планировщик

Менеджер памяти

GC

Ядро

Исполнитель

Интерпретатор

JIT компилятор

Профилятор

загрузка/cохранение

Данных и кода

Модель данных

Внешний интерфейс

Инициализация

Инкапсуляция — принцип модульности, система разделяется на модули, «внутренности» которых скрыты от остальных.

Среды управляемого исполнения — эксзоскелет вокруг пользовательского кода.

Цена модульности — накладные расходы при переходе от одного модуля к другому.

Модели данных: модели потоков, нитей, рефлексий, итд

Профилятор — постоянно профилирует(oh, really?). С помощью профилятора достигается локальный оптимальный баланс между скоростью работы и временем компиляции.

Менеджер памяти — выделяет память, обнаруживает живые объекты, собирает мусор(освобождает память).

Для каждого семейства сборщиков мусора — своя стратегия выделения памяти.

Планировщик — следит за компонентами системы и синхронизацией(основной вид синхронизации — исполнение большого числа программ на малом числе процессоров)

* Создание и разрушение нитей
* Распределение нагрузки между процессорами
* Приостановка и возобновление нитей
  + Для синхронизации управляемого кода
  + Для сборки мусора
  + Для синхронизации неуправляемого кода и сборки мусора

Интерпретатор — медленный, работает через внешний интерфейс

JIT — компилятор

Есть возможность сохранить промежуточное представление(Intermediate Representation)(часть работы JIT — компилятора) это ускорит последующие компиляции, однако создает уязвимости.

Один из методов борьбы с такими уязвимостями Trusted computing — все промежуточные данные подписываются цифровой подписью

Верификация

* Проверка формата класса
* Проверка данных класса
  + final классы и методы не перекрыты
  + Определен суперкласс
  + Формат пула констант
* Проверка кода класса
  + Размер стека, тип данных на стеке
  + Правильный тип локальной переменной при доступе
  + Число аргументов при вызове метода
  + Необходимое количество и тип аргументов инструкций
* Проверка ссылок
  + Наличие классов, методов и полей
  + Права доступа

Опр: Указатель — число, адрес ячейки памяти.

Опр: Ссылка — уникальный идентификатор какой- либо сущности.

Внешний интерфейс — позволяет написать методы, для тех случаев, когда невозможно целиком написать приложение на языке Java.

Например, стандартная библиотека класса Java не поддерживает платфоременные особенности или программную библиотеку.

Данный фреймворк позволяет нативному методу использовать Java объекты также, как Java код. Нативный метод может создавать Java объекты, а затем просматривать и использовать для выполнения своих задач. Нативный метод также может просматривать и использовать объекты, созданные кодом приложения Java.

* Вызов внешних неуправляемых функций
* Обратный вызов управляемого кода (callback) — передача внешней библиотеке указателя на функцию, которая будет обрабатывать события.
  + Два сопособа итераций:
    - внешняя итерация

for(i=0; i<=n; i++)  
a.next[i]

* + - внутренняя  
      a.interate (это и есть callback)
* Передача данных между управляемым и неуправляемым кодом (marshaling)
  + Копирование — эффективно для малых данных и неэффективно для больших
  + Обертывание и обратный вызов — решает безопасным способом абсолютно все задачи, но есть проблемы с эффективностью
  + Закрепление объектов (pinning) — запрет объектам перемещаться из памяти
  + Указатель на неуправляемую память

Типы команд

* Простые
  + Стековая арифметика
  + Доступ к локальным данным
  + Ветвления
* Объектные
  + Создание объектов
  + Доступ к полям
  + Вызов методов
  + Исключения

**3. Внешний интерфейс Java (JNI)**

* Внешний модуль должен использовать JNI
* Копирование для примитивных типов (параметры и результат)
* Обертывание и обратный вызов для объектов (globalRef, localRef)
  + Требует согласования с планировщиком
  + Потери времени при вызове
  + Потери времени при доступе
  + Возможна изоляция управляемых данных
  + Групповые операции с массивами
  + Усложнение внешнего модуля
* Абстрактный неуправляемый массив (DirectBuffer)

Ссылки передаются по значению → неуправляемый код не получит доступ на внутренние ресурсы машины .

Между быстро и безопасно разработчики Java выбрали безопасно.

**4. Внешний интерфейс CLI**

Common Language Infrastructure

Разработан Microsoft. Менее безопасный чем JNI. В C# и C++ есть «небезопасное» множество размером примерно с С

* Прямое управление раскладкой полей объекта
* Уменьшает гибкость системы
* Закрепление объекта в памяти и получение указателя
* Уменьшает гибкость менеджера памяти
* Потенциальная угроза безопасности
* Прямая работа с указателями из управляемого кода
* Потенциальная угроза безопасности
* pinvoke (.NET)
* Связывание произвольной внешней функции
* копирование объектов с изменением раскладки полей

Функции, реализованные в неуправляемых библиотеках DLL, могут вызываться из управляемого кода с помощью вызова неуправляемого кода P/Invoke. Если исходный код библиотеки DLL недоступен, вызов P/Invoke является единственным вариантом обеспечения взаимодействия. В отличие от других языков .NET в Visual C++ предусматривается альтернатива вызову P/Invoke.

Все совместимые языки компилируются в Common Intermediate Language(CIL), который является промежуточным языком, абстрагированным от платформенного «железа». Во время исполнения кода CIL будет перекомпилирован в машинный язык, соответствующий конкретному «железу» и ОС.

Промежуточное представление(IR) :

* Язык регистровой машины
  + На 45% меньше операций доступа к памяти
  + Возможность оптимизации на уровне IR
  + Используется в статических и оптимизирующих JIT компиляторах
* Язык стековой машины
  + На 25% более компактен
  + Сохраняет семантику исходного кода(проще восстановить исходный код)
  + Проще для интерпретации и шаблонной компиляции
* Используется в JBC(Java Beans?) и CIL

Оба языка полны по Тьюрингу(эквивалентны машине Тьюринга)

**5. Процесс верификации класса Java**

Правило Гослинга — размер стека и тип переменных не зависит от пути в графе управления.

Граф управления — все возможные варианты реализаци ветвлений в программе.

Граф потока данных — для каждой операции есть набор входных данных и набор выходных данных.

Верификация

* Проверка формата класса
* Проверка данных класса
  + final классы и методы не перекрыты
  + Определен суперкласс
  + Формат пула констант
* Проверка кода класса
  + Размер стека, тип данных на стеке
  + Правильный тип локальной переменной при доступе
  + Число аргументов при вызове метода
  + Необходимое количество и тип аргументов инструкций
* Проверка ссылок
  + Наличие классов, методов и полей
  + Права доступа

По стандарту проверка ссылок осуществляется лениво.

Алгоритмически задача верификации неразрешима за конечное время.

[https://blogs.oracle.com/vmrobot/entry/%D0%B2%D0%B5%D1%80%D0%B8%D1%84%D0%B8%D0%BA%D0%B0%D1%86%D0%B8%D1%8F\_%D0%B1%D0%B0%D0%B9%D1%82\_%D0%BA%D0%BE%D0%B4%D0%B0\_%D0%B2\_jvm](https://blogs.oracle.com/vmrobot/entry/верификация_байт_кода_в_jvm)

Верификатор хранит состояние локальных переменных и стека в каждой точке программы. Когда некоторая инструкция программы выполняется еще раз, сохраненное состояние объединяется с текущим, а разные типы заменяются общим супертипом.

Затем, если инструкция загружает переменную или значение из стека, ожидаемый тип сравнивается с действительным. Если действительный тип не может быть присвоен ожидаемому типу (т.е., не является тем же типом или потомком ожидаемого типа в дереве), класс считается невалидным.

Таким образом, верификатор пытался определить типы, имитируя выполнение программы. В худшем случае, сложность верификации была O(n \\* log2 t), где t — количество типов в дереве типов (включая классы Java), а n — количество инструкций.

**6. Интерпретация. Шаблонная JIT компиляция. Оптимизирующая JIT и AOT компиляция.**

Интерпретация - пооператорный (покомандный, построчный) анализ, обработка и тут же выполнение исходной программы или запроса (в отличие от компиляции, при которой программа транслируется без её выполнения).

Интерпретация работает по принципу конвейера, возникают конфликты, это приводит к простоям(чтобы уменьшить простои необходимо увеличить линейные участки кода).

В интерпретации поддерживается исполнение супер инструкций — характерных последовательностей инструкций.

Динамическая(Just in Time) компиляция

Компиляция IR в машинный код происходит непосредственно вовремя работы программы

При вызове незагруженного класса JIT компилятор выполняет частичную компиляцию, а затем загружает нужный класс(не выполняется для super-class).

Шаблонная компиляция — сборка бинарного кода из небольших фрагментов кода.

Оптимизирующая компиляция — используются различные методы получения более оптимального программного кода при сохранении его функциональных возможностей.

* Виды:

* Peephole-оптимизации рассматривают несколько соседних (в терминах одного из графов представления программы) инструкций, чтобы увидеть, можно ли с ними произвести какую-либо трансформацию с точки зрения цели оптимизации. В частности, они могут быть заменены одной инструкцией или более короткой последовательностью инструкций.
* Локальная оптимизация- рассматривается только информация одного базового блока за один шаг Так как в базовых блоках нет переходов потока управления, эти оптимизации требуют незначительного анализа (экономя время и снижая требования к памяти), но это также означает, что не сохраняется информация для следующего шага.
* Внутрипроцедурные оптимизации — глобальные оптимизации, выполняемые целиком в рамках единицы трансляции (например, функции или процедуры) При наличии глобальных переменных может быть затруднена.
* Оптимизации циклов
* Межпроцедурные оптимизации

Ahead of Time(AOT) — компиляция перед исполнением.

Это — статическая компиляция.

Экономит память, нет подвисаний во время исполнения программы.

Между сессиями часть runtime environment можно сохранить, это экономит память, время запуска программы.

Harmony Jet - наивный шаблонный компилятор.

Harmony Interp - наивный интерпретатор

**7. Организация памяти современного процессора (L1, L2, L3 Cache, TLB, Virtual memory)**

Процессор — быстрый, память — медленная. Чем быстрее память, тем она дороже.

Кэш — память — быстрая, но маленького объема. В нее надо помещать наиболее часто используемые слова.

Кэш базируется на принципе локальности:

* Временная — данные, которые были использованы скорее всего будут использованы еще раз.
* Пространственная — соседние с использованным элементом данные скорее всего будут использоваться

Если требуемого слова нет в кэше, оно подгружается из основной памяти вместе со соседними, вытесняя при этом часть загруженных в кэш слов. Один из принципов вытеснения — Least recently used — вытсенение самых старых данных из кэша.

Префетчинг — подвыборка данных, программа начинает их читать раньше, чем из

используют.

Хорошая статья:

<http://iproc.ru/parallel-programming/lection-7/2/>

N — ассоциативный кэш:

Память состоит из кусочков - «банков». Кэш состоит из отдельных ячеек — слов. N — количество ячеек памяти, сопоставляющихся одному банку. При увеличении N увеличивается число проводов => увеличивается их длина. Нужен компромисс.

Несколько уровней кэша, работающих с разной скоростью

* L1 — 3
* L2 — 9 - 15
* Память — 30 - 180

Виртуальный ресурс(виртуальная память?) — абстракция, создающая иллюзию владения одним процессом всего адресного пространства

Виртуальная память — метод, отображающий адреса памяти, используемые программой, названные виртуальными адресами в физические адреса в памяти компьютера. Для программы адресное пространство выглядит как доступное и непрерывное адресное пространство, либо как набор непрерывных сегментов

Страничная организация виртуальной памяти: Оперативная память делится на страницы: области памяти фиксированной длины (например, 4096 байт), которые являются минимальной единицей выделяемой памяти (то есть даже запрос на 1 байт от приложения приведёт к выделению ему страницы памяти). Исполняемый процессором пользовательский поток обращается к памяти с помощью адреса виртуальной памяти, который делится на номер страницы и смещение внутри страницы. Процессор преобразует номер виртуальной страницы в адрес соответствующей ей физической страницы при помощи буфера ассоциативной трансляции (TLB). Если ему не удалось это сделать, то требуется дозаполнение буфера путём обращения к таблице страниц (так называемый Page Walk), что может сделать либо сам процессор, либо операционная система (в зависимости от архитектуры). Если страница была выгружена из оперативной памяти, то операционная система подкачивает страницу с жёсткого диска в ходе обработки события Page fault. При запросе на выделение памяти операционная система может «сбросить» на жёсткий диск страницы, к которым давно не было обращений. Критические данные (например, код запущенных и работающих программ, код и память ядра системы) обычно находятся в оперативной памяти (исключения существуют, однако они не касаются тех частей, которые отвечают за обработку аппаратных прерываний, работу с таблицей страниц и использование файла подкачки).

TLB(translation lookaside buffer) - это специализированный кэш центрального процессора, используемый для ускорения трансляции адреса виртуальной памяти в адрес физической памяти. TLB используется всеми современными процессорами с поддержкой страничной организации памяти. TLB содержит фиксированный набор записей (от 8 до 4096) и является ассоциативной памятью(Особый вид машинной памяти, используемый в приложениях очень быстрого поиска). Каждая запись содержит соответствие адреса страницы виртуальной памяти адресу физической памяти. Если адрес отсутствует в TLB, процессор обходит таблицы страниц и сохраняет полученный адрес в TLB, что занимает в 10—60 раз больше времени, чем получение адреса из записи, уже закэшированной TLB. Вероятность промаха TLB невысока и составляет в среднем от 0,01 % до 1 %.

**8. Представление в памяти объекта и класса. Хеш код объекта, 2 битная упаковка, однобитная упаковка слова состояния монитора**

**9. Реализация вызовов через интерфейс.**

10.