ОС Фантом Internals

JPoint 2022

Что такое ОС Фантом

- Не основанная на Unix ОС, которая базируется на двух принципах. Всё – объект, объекты живут в глобальной персистентной оперативной памяти.
- Сканирование памяти невозможно, поэтому код имеет доступ строго к тем объектам, на которые у него есть указатели.
- Проще всего представить себе среду исполнения Фантом как сервер приложений (типа JVM/.net), который живёт вечно (гарантированно переживает выключения компьютера) и в силу этого факта практически не нуждается в понятиях «файл» или «база данных».

Глобальная память, полная защита

- Общее адресное пространство.
- Лёгкая и дешёвая модель взаимодействия компонент = простота разработки.
- Защита доступа на уровне объекта (переменной), а не программы целиком.
- Защита программы от библиотек.

Надежность

- Состояние всей ОС сохраняется не только при штатном останове, но и при внезапной перезагрузке или отказе оборудования.
- Возможна модель быстрых рестартов при временном или частичном отказе оборудования.
- Общий сбой системы, полный сбой части оперативной памяти, полный сбой части процессоров.
- In-memory рестарт в разработке.

Защита

- Программа не может получить доступ к объекту, если явным образом не получила ссылку на него от другой программы.
- Даже в случае успешной загрузки вируса в систему он не получит никаких данных кроме тех, что принёс с собой.
- Атака путём подмены системных библиотек сложна и не обеспечивает доступа к произвольным данным.
- Возможна цифровая подпись и версионирование классов.
- Возможно обеспечение индивидуальной среды работы для отдельных программ на одной машине.

Структура системы

- Ядро ОС обеспечивает персистентную оперативную память
- Виртуальная машина и сборщик мусора обеспечивают защищенное исполненние прикладного кода.
- Набор прикладных библиотек обеспечивает доступ к сервисам ядра (сеть, графика, etc)
- Встроенная в ядро подсистема управления рестартом реализует консистентное востановление состояния после отказа оборудования.

Persistent memory

Application1 state (daemon)

Class1 code

obj

obj

Socket

Class2 code

Unix emulation instance

Data segment

Native code

Thread

Application2 state (document)

obj

Window

obj

Class3 code

Binary object containing videostream (seen as file from Unix code)

Kernel

Network stack

Internal classes methods

Persistent Pager

Virtual Machine

Unix syscalls

Video/3D engine

Drivers

Нужен сборщик мусора

- Очевидно, в такой среде необходим сборщик мусора. Но. Обычный GC не подходит, так как рассчитан на типовой размер адресного пространства процесса.
- Нужен рассчитанный на адресное пространство всей системы, включая неактивные приложения – на порядки больше. Фактически, на размер порядка размера диска. Со скоростью доступа к объектам характерной не для оперативной памяти, а для диска.
- С учётом запаса на 20 лет тому вперед – это сотни терабайт, если не петабайты.

Есть хорошие новости

- Есть снапшот всего адресного пространства, с ним можно работать не спеша. Это позволяет собрать «сложный» мусор.
- Но система это время должна прожить. Значит, нужен ещё один GC - быстрый, но, возможно, неидеальный.
- Их нужно синхронизировать.

Базовый сборщик мусора

- Традиционный mark & sweep
- Работает на копии адресного пространства систсемы. Копия всегда устарела, но то, что мусор вчера – мусор и завтра.
- Не трогает зону молодого поколения

Быстрый сборщик мусора

- Работает онлайн
- Основан на счётчиках ссылок
- Не может освободить циклические структуры
- Работает только в зоне молодого поколения, чтобы не пересечься с базовым сборщиком

Race condition

- Тред 1 только что прочитал ссылку на объект А из объекта Б
- Тред 2 стёр эту ссылку (записал в этот слот другую) и уменьшил счётчик на 1.
- Счётчик обнулился и объект А был уничтожен
- Тред 1 увеличил счётчик объекта А – уже уничтоженного только что – доступ произошёл к уничтоженному или неверному объекту

Stop the race

- Проблема решается, если уничтожение объектов после обнуления счётчиков отложить. Требуемое условие – чтобы все инструкции виртуальной машины, которые исполнялись в момент обнуления счётчика, закончились.
- Это требование трудно исполнить, но есть другое. Перед выполнением снапшота система приостанавливает все треды, причём именно в точках МЕЖДУ инструкциями виртуальной машины.
- Именно в этот момент можно безопасно освободить все объекты с нулевыми счётчиками ссылок.

Верхи не хотят...

- Есть одна проблема – с точки зрения системы снапшотов созданный, использованный и удалённый объект в 10 терабайт размерами – это модифицированный участок оперативной памяти, и при создании следующего снапшота его надо записать.
- Между тем практической ценности он не имеет, и можно бы его не записывать. В этом месте возможна оптимизация. Нельзя не записывать в снапшот заголовки объектов, иначе сломается система управления памятью. Но можно не записывать тела.
- Идеальный вариант – перед записью снапшота склеить вместе все свободные последовательные объекты, и записать в снапшот только первую страницу этой зоны памяти.

Вспомнить всё

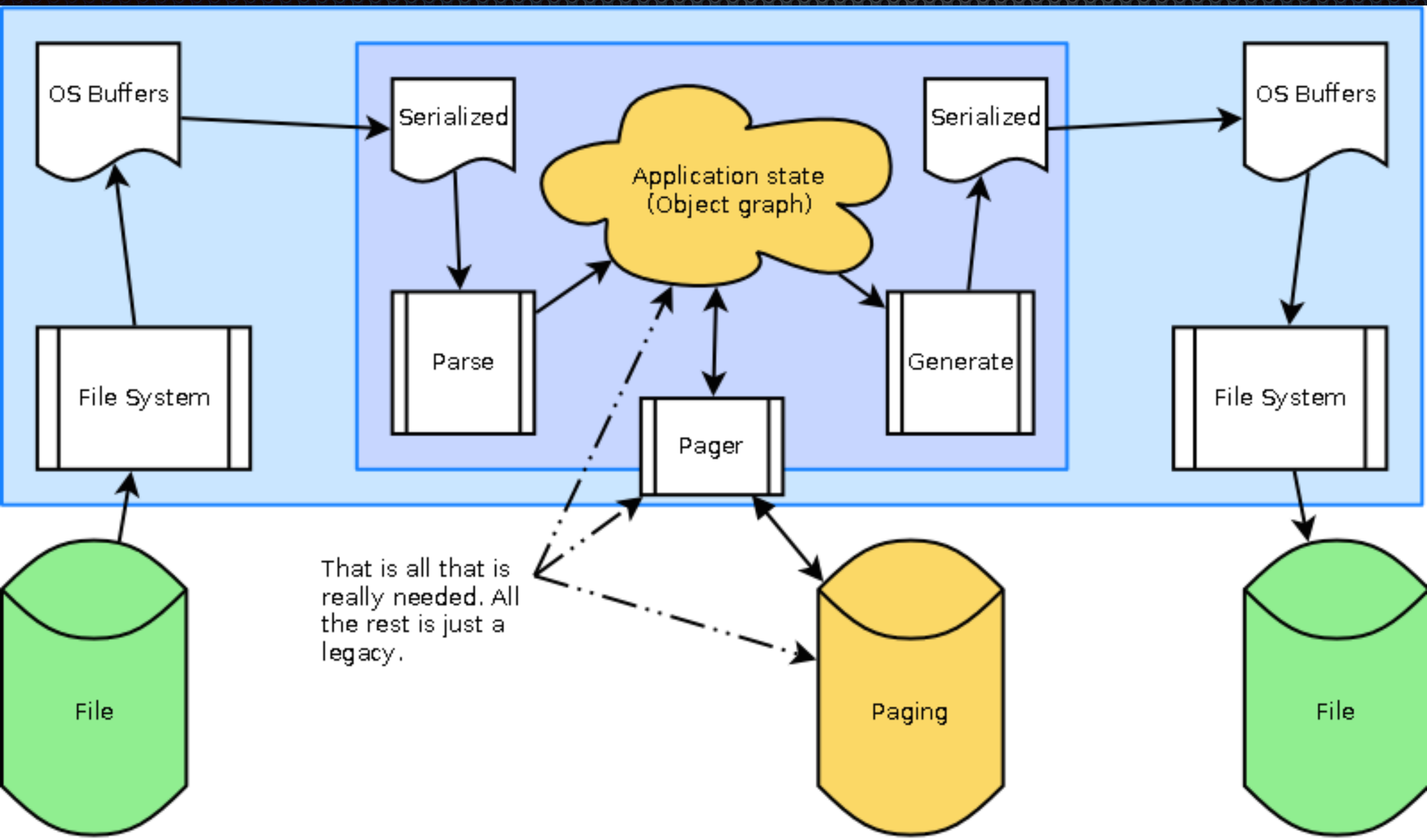
- Нет хорошего ответа на вопрос что делать, если мы потеряли один блок на диске – по хорошему большой сборщик мусора должен обладать свойствами fsck – проверять целостность объектного пространства (это понятно, как делать и частично сделано) и восстанавливать его, возможно, перенося частично повреждённые объекты в lost & found – это пока серая зона, реализация неочевидна.

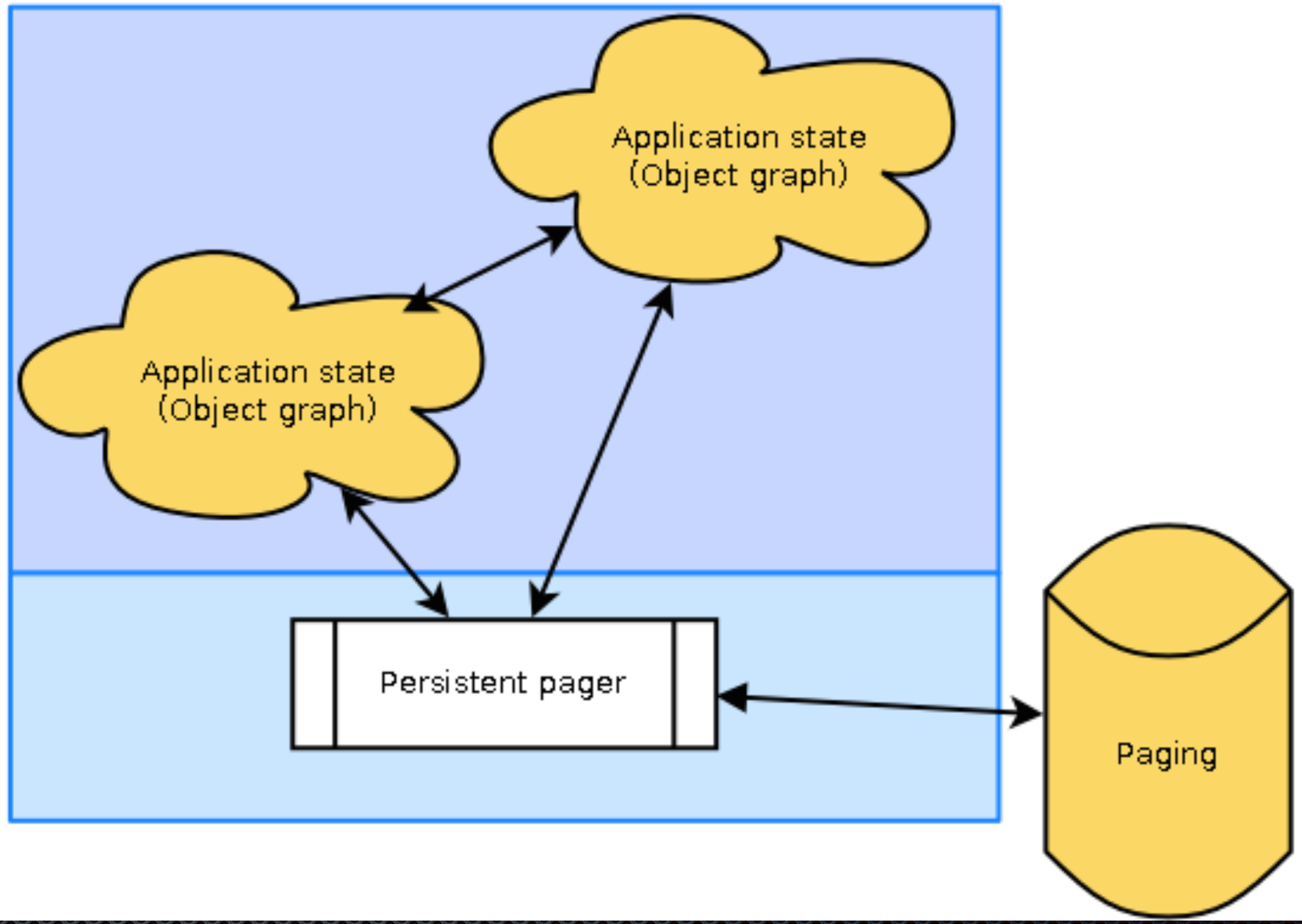
Вопросы

- https://phantomdox.readthedocs.io/en/latest/
- Дмитрий Завалишин, dz@dz.ru
- DZ Systems group, http://dzsystems.com:
  - Digital Zone, http://dz.ru
  - E-legion, http://e-legion.com
  - Aprentis, http://aprentis.ru

Независимость

- Не используется код ОС Linux или какой-либо иной ОС.
- Код ядра крайне компактен и обозрим.
- Есть собственный компилятор и язык для разработки прикладного ПО.


OS Buffers
Serialized
Application state
(Object graph)
Serialized
OS Buffers
Parse
Generate
File System
File System
Pager
That is all that is
really needed. All
the rest is just a
legacy.
File
Paging
File


Application state
(Object graph)
Application state
(Object graph)
Persistent pager
Paging

Традиционные ВМ

- Переход от компилируемых языков программирования таких как C и C++ к языкам на базе байткодных виртуальных машин (языки с управляемой - managed - памятью) повысил защищённость программ как от ошибок в самих программах, так и от взлома.
- Тем не менее, в этом направлении сделано не всё возможное.

Традиционные ВМ

- Native код вообще ничем не ограничен и может иметь доступ к произвольной части памяти
- Хранение кода и статических данных опирается на файлы и традиционную систему защиты доступа, промежуточные данные зачастую тоже хранятся в файлах
- Возвращаемое значение зависит от вызываемой функции и, в принципе, может иметь неверный тип или вовсе отсутствовать.
- Существуют функции - методы без параметра this
- Native функции отличимы на уровне исходного кода и реализации
- Некоторые типы данных (int, double) не являются классами и поддерживаются специальным образом

Задача

- Операционная система Фантом ориентирована на исполнение в общем адресном пространстве множества приложений, которые не могут доверять друг другу, но, возможно, будут вызывать методы друг друга. Хотелось бы полностью исключить хотя бы катастрофические последствия таких вызовов, в случае возникновения рассогласования между вызывающей и вызываемой функцией.

Унификация вызова

- Вызов в Фантоме унифицирован абсолютно — это всегда вызов метода (есть this и есть класс), и всегда возвращается значение, которое для void метода равно null и явно уничтожается вызывающим кодом. Это гарантирует, что какая бы ошибка вызова не случилась, что бы не подвернулось в качестве предмета вызова, протокол вызова и возврата будет соблюдён.

Унификация представления

- Есть отличие и в работе с целыми. Ява выделяет их в отдельную категорию типов, отличную от объектных, “классовых” типов — java.lang.Integer и int — разные вещи в Яве. Компайлер иногда удачно скрывает этот факт, но внутри они различаются. Фантом и здесь идёт в сторону максимализма. Целое — честный объект. Его можно вытащить на целочисленный стек и там посчитать в “необъектной”, бинарной форме, но он вернётся в форму объекта будучи присвоен переменной или передан в параметре. Это, кстати, тоже вытекает из требования униформности протокола вызова метода — методы, возвращающие целое и объект по протоколу тождественны. (То же самое, очевидно, относится и к другим «интегральным» типам — long, float, double.)

Унификация класов

- Протокол подключения того, что в Яве называется native методы. В Фантоме это «системные вызовы», и, опять же, на уровне вызова метода они ничем не отличимы от обычного “честного” метода. (Код такого метода содержит специальную инструкцию для “ухода” в ядро ОС, но “снаружи” метода это не видно. Это, в частности, позволяет наследовать и оверрайдить такие методы традиционным путём, через замену VMT.)

- Есть возможность ограничить такому методу доступ к памяти

Без файлов

- Существует механизм, который позволяет загрузить константные данные, необходимые для работы программы - например, изображения для графической подсистемы или векторные данные для картографии - напрямую в переменные. Персистентная программа будет запущена сразу с нужными данными в переменных, и к этим данным можно будет обратится напрямую, без загрузки их из файла. При этом размер таких данных практически не ограничен.
- Аналогично промежуточные значения могут храниться точно так же просто в переменных - поскольку в программах для ОС Фантом содержание переменных не пропадает при останове операционной системы.

Без файлов

- Такой подход снижает трудоёмкость написания программ и уменьшает риски как случайных ошибок, так и сознательного нанесения ущерба. Временные файлы физически доступны извне программы и могут быть случайно или по недоброжеланию повреждены или уничтожены. Персистентные переменные программы абсолютно недоступны извне.

Структура стека и оптимизация

- Фактически, стек в программах для Фантом отсутствует - впрочем, аналогично он отстутствует и в большинстве виртуальных машин. Вместо стека применяется связный список записей активации, по одной на функцию, в каждой из которых находится несколько рабочих стеков для функций.
- Это проистекает из желания защитить функции друг от друга и избежать, к примеру, возможности нарушить работу программы в целом, исполнив избыточное количество операций pop, “съев” таким образом не только часть стека, относящуюся к текущей функции, но и зайдя на “территорию” функции предыдущей.

Оптимизация

- Стек в виде списка гарантирует нас от этой проблемы, но снижает производительность программ, так как аллокация и удаление фреймов (записей активации) усложняется.
- В обычных виртуальных машинах, как правило, с этим ничего нельзя поделать. Но если виртуальная машина, как это сделано в ОС Фантом, есть часть ядра, возникают дополнительные возможности. Мы рассматриваем возможность реализации режима частично защищённого стека, когда проверки прав доступа на части стека будут либо возложены на подсистему виртуальной памяти - и тогда стек каждой очередной функции должен начинаться с новой 4-килобайтной страницы, или же за счёт специальной аппаратуры процессора. В x86 для этого можно применять сегментную адресацию, а Эльбрус и вовсе обеспечивает на аппаратном уровне полную защиту стека, полностью соответствующую ожиданиям виртуальной машины Фантом.

Вопросы

- https://phantomdox.readthedocs.io/en/latest/
- Дмитрий Завалишин, dz@dz.ru
- DZ Systems group, http://dzsystems.com:
  - Digital Zone, http://dz.ru
  - E-legion, http://e-legion.com
  - Aprentis, http://aprentis.ru