ЗАЩИЩЕННЫЕ ИНФОРМАЦИОННЫЕ СИСТЕМЫ

Проф. Борис Бабаян

2003 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Почему пишется эта работа

В связи с громадным распространением в последнее время информационных систем на все области человеческой деятельности и создавшейся большой зависимости всей жизни общества от правильной их работы, возросла до критического уровня важность защиты этих систем от пагубных последствий ошибок в программах и внешних атак злоумышленников.

Еще в начале развития вычислительной техники, до начала 80х годов во многих западных университетах и коммерческих фирмах велись интенсивные исследования в этой области, создавались экспериментальные и опытные образцы систем нацеленных на решение этих проблем. Позже в данной работе будет приведен исторический обзор и анализ наиболее значительных из них.

Эта проблема привлекла внимание и фирмы Интел, этого признанного лидера в области микропроцессоров. В начале 80х годов была разработана система 432 фирмы Интел, одной из главных целей которой была разработка системы защищённого программирования.

К сожалению, приходится констатировать, что даже этот лидер вычислительных систем не смог найти правильного решения проблемы. Эта система будет более детально рассмотрена в данной работе, и будут отмечены её основные технические промахи.

Создание неудачно завершившейся системы 432, положила конец всем работам по созданию защищённых систем на Западе. По всей видимости, 432 система Интел убедила Западных разработчиков в бесперспективности этого направления.

До того как неудача этой системы стала всем очевидной, в Советском Союзе ставился вопрос, в привычном стиле для тех времён, о копировании её на наших предприятиях. Автор данной работы выступил с резкой критикой этих намерений и самой системы Интел 432, показывая её техническое несовершенство.

К счастью, в это время наш коллектив уже имел за плечами успешный опыт разработки и эксплуатации системы, в которой, в частности, была полностью и успешно решена проблема защищённости вычислений. Речь идёт о вычислительном комплексе Эльбрус 1, который заработал в 1978 году и находился в широкой эксплуатации в стране. Завершалась разработка второго поколения системы – Эльбрус 2, выпущенного в 1985 году. Архитектура микропроцессоров Эльбрус 3М и Е2К, включает в себя эту технологию, адаптированную для современного окружения.

Защищённая технология Эльбруса основана на использовании контроля типов на всех уровнях системы: в аппаратуре, в языке и в операционной системе. Сама по себе идея не нова, существо нашей работы в большом количестве деталей, позволивших на базе

последовательного проведения этой идеи, создать эффективно работающую коммерческую систему, не имеющую аналогов в мире до настоящего времени.

Созданная технология убедительно продемонстрировала её важнейшие свойства.

Отладка программ, в том числе больших программных систем, значительно упростилась, т.к. большинство трудно обнаруживаемых, семантических ошибок, превратились в формальные ошибки, автоматически обнаруживаемые системой во время исполнения программ.

По мнению большинства пользователей и независимых экспертов, специально исследовавших систему, отладка программ на ней требует на порядок (в десять раз) меньше времени.

Качество программ, разработанных на этой системе значительно выше, т. к. значительно сокращается вероятность не обнаруженных ошибок в программах переданных пользователям в эксплуатацию.

Наконец, её важнейшим свойством является защита от опасности вирусов и злонамеренных атак. Черта особенно важная для современных условий.

Основные результаты этих работ не были достаточно полно освещены в отечественной печати, а зарубежные читатели и разработчики информационных систем совсем не знакомы с ними. Эта работа как раз и призвана восполнить этот пробел. Цель её осветить на достаточно детальном уровне существо и результаты этой уникальной по своему характеру работы и сделать их достоянием широкого компьютерного сообщества.

О чём эта работа

В данном разделе мы обсудим понятие защищённой информационной системы и причины, которые могут нарушить её правильную работу.

Существуют несколько причин, которые могут нарушить правильную работу системы и от которых желательно защитить её работу.

Это, прежде всего, сбои и отказы оборудования. Защищённая система должна надёжно их обнаруживать и ликвидировать их последствия на работу системы. В машинах серии Эльбрус реализована надёжная защита от этих проблем, основанная на полном аппаратном контроле, резервировании аппаратуры и аппаратно – программной системы восстановления. Эта технология позволяет обнаруживать и ликвидировать последствия всех одиночных сбоев и отказов, обеспечивая, тем самым, полную достоверность вычислений.

Современное оборудование обладает достаточно высокой надёжностью, поэтому в настоящее время применять упомянутою технологию имеет смысл лишь в особо критических случаях. Описание этой технологии не включено в данную работу и может быть предметом отдельного изложения. В дальнейшем в данной работе мы будем предполагать надёжную работу оборудования.

Подавляющее большинство современных машин работают в компьютерных сетях через линии связи. Линии связи могут стать другой причиной неправильной работы системы. При этом следует бороться как со сбоями и отказами самих линий связи, так и злонамеренными атаками с использованием линий связи. В настоящее время борьба со всеми этими явлениями достаточно хорошо проработана. Эти технологии не являются областью разработок команды Эльбрус. Поэтому в данной работе мы так же будем предполагать надёжную работу линий, защищённую от злонамеренных атак. Мы будем предполагать, что информация, передаваемая по линии, доходит до адресата без искажений и адресат получает достоверную информацию о том, кто её передал.

Таким образом, мы будем предполагать правильную и надёжную работу аппаратуры и правильную работу линий, защищённую от внешних атак.

Чтобы данные вычисления были полностью защищены, нам ещё необходимо сделать крайне важный и, быть может, наиболее трудный шаг - защитить их от всякого влияния злонамеренных атак и ошибок других вычислений на данной или любой другой машине, работающей в сети. Т.е. необходимо защитить вычисления от своих программных ошибок и\или злых намерений других программистов. Действительно, если вычисления происходят на правильно работающем оборудовании, и они изолированы от разрушительного влияния со стороны каких-либо других внешних вычислений на той же, или какой-либо другой машине, то можно считать, что система обеспечивает защищённость вычислений.

Именно в этой области существует большой пробел в современных информационных системах. Изложению технологии Эльбруса, обеспечивающей решение этой проблемы, и будет посвящена данная работа.

БАЗОВЫЕ ИДЕИ

Что надо защищать в информационной системе

Основными составными частями любых вычислений и любой информационной системы являются обрабатывающие устройства и памяти всех уровней.

Обрабатывающие устройства, работающие по программе данных вычислений, не требуют специальной защиты от внешних воздействий. Проблему здесь представляют ошибки в самой программе вычислений, от которых следует защищаться, и которые будут предметом наших забот в данной работе.

В памяти, напротив, проблемой является её защита от внешних воздействий.

Таким образом, мы должны защитить память данных вычислений от внешних воздействий и побороться с ошибками в собственной программе.

Часто внешние атаки на определённую машину или размножающиеся атаки, какими являются вирусы, используют ошибки в системных программах для достижения своих целей.

Обе эти проблемы (собственные ошибки и внешние атаки) в технологии Эльбрус решены единым методом на достаточно фундаментальном уровне.

Основной стратегический подход при создании защищённой технологии Эльбрус

Дело в том, что концептуальная модель вычислений, используемая во всех современных языках программирования высокого уровня, представляет достаточно надёжную основу для создания высоко защищённых информационных систем. По существу она совсем не страдает описанными выше недостатками. Все проблемы появляются при реализации этих концепций. Кроме того, для полного решения проблемы на подобных же идеях должны быть разработаны операционная система и аппаратура.

Но даже и при разработке языков, типичный ход событий выглядит следующим образом. Начиная с надёжных позиций, обеспечивающих создание высоко защищённого языка, в процессе более детальной проработки, обычно в целях повышения эффективности, эти простые и надёжные исходные положения размываются, модифицируются, приводя в результате, конечно к более эффективному, но уже совершенно не надёжному языку.

По крайней мере, две причины обуславливают такую ситуацию.

Во-первых, в прошлом, когда создавались все эти системы, оборудование было медленное и громоздкое, и задача достижения высокой эффективности имела высший приоритет.

С другой стороны, проблема защищённости совсем не была такой острой. Не было компьютерных сетей, через которые в настоящее время ведётся большинство атак. Значительно реже происходил обмен программами, и т. д.

Во-вторых, упомянутые выше надёжные концептуальные модели, даже в качестве только исходных позиций, использовались лишь при разработке языков программирования. Ни при разработке операционных систем, ни, тем более, при разработке архитектуры вычислительных машин, соображения защищённости не присутствовали даже в таком виде.

Технология Эльбрус радикально отличается от описанного выше традиционного подхода.

Базовые решения, развитые в языках, эффективно поддержаны в аппаратуре и в операционной системе Эльбрус. По существу в Эльбрусе это единые решения, реализованные в современных языках, операционной системе и аппаратуре.

Технология Эльбрус была разработана в 70е годы, но актуальность её неизмеримо возросла в последние годы, когда незащищённость информационных систем, можно сказать без

преувеличения, создала критическую ситуацию для развития всего индустриального сообщества.

Реализация этой технологии требует, разумеется, некоторых ресурсов и ведёт к некоторому сравнительно небольшому снижению производительности, однако, в настоящее время, это представляется вполне разумной ценой за столь актуальное свойство.

Важной чертой обеспечения защищённости в технологии Эльбрус является тот факт, что она обеспечивается простыми решениями на базовом уровне основ информационной системы. В результате полное выполнение требований защищённости либо оказывается очевидной, либо легко доказывается. При этом остаётся позаботиться, чтобы не внести ошибки при реализации небольшого ядра, обеспечивающего реализацию этих простых решений.

Это выгодно отличает разработанную технологию от существующих в настоящее время методов, когда в системе, не обеспечивающей защищённости по своим базовым концепциям, начинает «штопаться» огромное, можно сказать, не предсказуемое число «дыр». При этом ни когда нет уверенности, что найдены все «дыры» и/или не допущены ошибки в громадном числе «штопок».

Базовые идеи защищённых информационных систем

Общая структура

В соответствии с обсуждённым выше планом, в этом и следующих разделах будет изложена базовая идея, являющаяся основой защищённых вычислений во всех языках высокого уровня. Здесь намеренно не будут рассматриваться проблемы реализации, так как, очень часто, именно они ответственны за пагубное для защищённости искажение базовых идей.

В этих разделах изложение не будет слишком детальным. Но оно будет достаточно детальным, чтобы стали ясными основные идеи. Это изложение будет охватывать не только языки, но и всю систему в целом, включая ОС и аппаратуру. Забегая вперёд, отметим, что именно реализация этих идей в аппаратуре, т. е. аппаратная поддержка защищённых вычислений, позволила создать эффективную систему, обеспечивающую защищённые вычисления в целом.

Однако проблемам реализации будут посвящены последующие разделы.

Сейчас же приступим к изложению базовых идей.

Как уже отмечалось выше, модель информационной системы можно себе представить состоящей из

- обрабатывающего устройства (ОУ), выполняющий все операции и хранящий некоторый объём данных, которые непосредственно используются и/или предположительно будут использованы в качестве аргументов операций и

- информационного пространства (ИП).

Намеренно постараемся в этом изложении не использовать для этой компоненты информационной системы слово «память», так как это понятие относится уже к реализации и, более того, именно подмена более абстрактного понятия информационного пространства реализационным понятием «память» и введением некоторых оптимизаций часто бывает ответственно за появление дефектов в защищённости системы.

В описываемой «идеальной» в смысле защищённости системе все вычисления разбиваются на фрагменты или вычислительные модули (ВМ). Смысл вводимого здесь термина «вычислительный модуль» несколько отличается от общепринятого, и означает наименьший объём вычислений, защищённый от других, ему подобных. Это наименьшая единица защищённости, в том смысле, что различные части одного и того же модуля уже не защищены друг от друга. Более точно это понятие будет определено позже (стр.17), но по смыслу это вычисления в текущей процедуре.

Соотношение между ИП, ОУ и ВМ можно представить следующим образом.

ВМ, так же как и обрабатываемые данные, размещается в ИП.

Информационная система устроена таким образом, что ОУ работает под управлением ВМ, и в процессе работы он может обращаться к ИП, считывать из него информацию во внутренние ячейки ОУ, обрабатывать их, вычисляя новые значения и различным образом менять ИП.

Рассмотрим более подробно базовые операции, которые может выполнять ОУ в ИП.

Создание нового узла

В репертуаре операций, которые может выполнять ОУ, предусмотрена операция создания узла ИП произвольного объёма для хранения данных. В результате в ИП создаётся новый узел заданного объёма, а в одной из ячеек ОУ появляется ссылка на него. Этот важный момент следует обсудить более подробно. Несколько слов о новом узле и о ссылке.

Как уже было отмечено, здесь не будет обсуждаться механизм создания нового объекта, где он расположен, что такое объём узла и т. д. Всё это реализация, о которой позже. Отметим только его важнейшие семантические свойства.

В момент появления новый узел не доступен ни кому, кроме данного ОУ. Доступ к нему возможен только через новую ссылку. Новый узел «пуст», т. е. не содержит информации.

Наилучший вариант «пустой» информации – это такая информация, которая вызывает диагностическое прерывание, при попытке использовать её в вычислениях. Во всех случаях, «пустой» узел не должен содержать старых ссылок, присутствие которых в новом узле может явиться результатом дефекта реализации. Наличие таких ссылок открыло бы доступ к информации, не принадлежащей к текущим вычислениям, что по существу является нарушением защищённости.

Ссылка

Теперь несколько слов о ссылке. Ссылка – это то, что в программистской практике называется POINTER, REFERENCE, УКАЗАТЕЛЬ и т. д.

Ссылка описывает единичный элемент данных или их агрегат и содержит разрешённые права доступа к нему. Она не только описывает данные, но и содержит всю необходимую информацию достаточную для фактического доступа к ним.

Обсудим теперь те операции, которые можно выполнять над ссылкой и те свойства, которым она должна удовлетворять.

Через ссылку на вновь созданный узел можно читать и изменять информацию, находящуюся в этом узле, в любой его части, можно, так же, использовать эту информацию в качестве кода для исполнения программы.

Если какой либо ВМ обращается к некоторым данным с помощью ссылки, то система не должна проверять имеет ли этот ВМ право обращаться к этим данным. Если ВМ может использовать определённую ссылку на некоторые данные, то это означает, что он имеет право обращаться к этим данным.

Это обстоятельство является важнейшим свойством ссылки.

Оно является следствием того факта, что программа ВМ не может произвольно создавать ссылки, они появляются в распоряжении программы ВМ лишь в результате создания данным ВМ новых данных или при передачи ссылки данному ВМ снаружи.

Однако в ОУ предусмотрены операции, которые позволяют ограничить этот доступ по характеру возможных операций и по области доступа. Это даёт возможность программе текущего ВМ передать в другие вычисления возможность доступа в созданный узел, но с определёнными ограничениями.

Для этого специальными операциями из произвольной ссылки можно создать новую ссылку на тот же узел, но с ограничением прав на возможный набор операций, которые можно выполнять с данной ссылкой или на область доступа в узел.

В качестве примера, рассмотрим три базовых права: читать, писать и исполнять информацию, на которую указывает ссылка. Эти права доступа для указателей обычно включают языки программирования.

Если ссылка разрешает чтение, то ограничение исполнения самого кода программы не имеет особого смысла, т. к. информацию можно сначала считать в другое место и затем оттуда её исполнить.

Однако работающая программа может модифицировать данные (например, глобальные) и запуск такой программы может модифицировать эти данные. Поэтому исполнение программы эквивалентно выполнению записи. В языках этот тип ссылки не разрешает исполнение.

Если же ссылка чтение запрещает, то малый смысл имеет разрешение записи. В этом случае практичнее использовать ссылку на процедуру, выполняющую запись переданного ей параметра в узел.

В результате из 8 ми в принципе мыслимых вариантов языки используют обычно только три типа ссылок:

- разрешены операции записи и считывания (ссылка на данные)
- разрешено только считывание (ссылка на константу)
- разрешено только исполнение (ссылка на функцию)

Эта классификация приведена здесь лишь в качестве примера. Вполне возможны и другие комбинации прав.

Следует отметить, что ссылка на константу запрещает запись только в те данные, на которые она указывает непосредственно, но не на все данные достижимые через эту ссылку. Если, например, через константную ссылку считать другую ссылку, не содержащую запрета записи, то можно модифицировать данные, на которые указывает считанная ссылка.

В принципе можно ввести ещё один тип ссылки (полная константа), который полностью запрещает модифицировать любую информацию прямо или косвенно доступную через этот тип ссылки. Для этого аппаратура должна превращать в полную константную ссылку любую ссылку, считанную через полную константную ссылку.

Другим способом ограничения доступа по ссылке является создание ссылки на подмножество узла. Если, например, в узле хранится массив, то можно получить новую ссылку на подмассив или на элемент массива. Если узел – структура, то новая ссылка может обеспечить доступ к элементу структуры и т. д.

Существует, разумеется, операция пересылки ссылки.

Приведём теперь важнейшее правило работы со ссылкой, выполнение которого в значительной степени ответственно за обеспечение защищённости информационной системы в целом.

Система должна обеспечивать контроль над тем, чтобы в операциях, предусматривающих использование в качестве аргументов ссылки, не были использованы данные других типов (целые, вещественные и т. д.), а в операциях с аргументами других типов ссылка не могла быть модифицирована.

Другими словами, при работе со ссылками должен быть обеспечен строгий контроль типов.

В языках высокого уровня этот контроль обеспечивает компилятор во время трансляции программы, так как в существующих машинах аппаратура не обеспечивает его во время исполнения программы.

Для того чтобы сделать статический контроль возможным в языки вводятся ограничения – тип данных, которые могут храниться в переменной, закрепляется за этой переменной на всё время её жизни.

Однако, как показывает практика, это неудовлетворительно, так как делает язык не вполне универсальным и понижает эффективность некоторых алгоритмов. На таком языке затруднено написание таких программ, как ОС, которая работает с данными пользователя, тип которых неизвестен во время трансляции ОС. Затруднено написание и программ обработки данных, считанных из сети, структура которых так же неизвестна при трансляции. Именно по этому в массовых языках, таких, как C и C++, в реализации контроль типов нарушается.

Для решения этой проблемы нужна поддержка аппаратуры, так, чтобы даже ассемблер машины не мог нарушить защищённость программ и системы.

Уничтожение узла

Наконец, надо уметь уничтожать созданный узел. В некоторых системах это делает сама система, если на какой-либо узел не осталось ссылок. В других системах это делает сама программа. Здесь важно отметить общее свойство операции уничтожения узла из программы. Если на уничтоженный узел остались ссылки, то попытка обращения через эти «зависшие» ссылки должна вызывать диагностическое прерывание.

Контекст

Выше было отмечено, что ОУ принадлежат некоторые данные, подлежащие обработке по системе команд.

В реальных системах эти данные физически располагаются в ОУ и «сбрасываются» в ИП только когда ОУ на время прерывает вычисления текущего ВМ и принимается за обработку другого ВМ. Однако это можно рассматривать как оптимизацию более простой и прозрачной организации, когда все эти данные постоянно располагаются в ИП и имеют имена в этом пространстве.

Отличие данных принадлежащих ОУ от других данных в ИП заключается в следующем.

Обычные данные в ИП доступны только по имени в ИП, т.е. через ссылку. Данные же принадлежащие к ОУ доступны из программного кода по статическим именам, таким, например, как «номер регистра общего назначения», «номер спецрегистра».

Пользуясь статическими именами, программный код может использовать ссылку, находящуюся в ОУ, для считывания данных из ИП. Если считанные данные содержат ссылку, то процедуру считывания можно повторить. Таким образом, можно добраться до любой информации из ИП в принципе доступной для данного ВМ.

Множество всех этих данных доступных для вычислений по программе текущего ВМ будем называть КОНТЕКСТом этих вычислений. Содержание КОНТЕКСТа может меняться в процессе вычислений, так как вычисления могут создавать новые узлы и уничтожать старые.

Единую ссылку, через которую можно получить доступ ко всей совокупности данных, принадлежащих ОУ во время выполнения текущего ВМ, будем называть КОНТЕКСТНОЙ ССЫЛКОЙ (КС) данного ВМ.

Несколько упрощённо дело можно представить следующим образом. Имена в программном коде – это индексы или селекторы, которые используются для индексации КС. В результате происходит обращение к данным, принадлежащим к ОУ.

Как можно понять из сказанного выше, с помощью КС можно добраться до любой информации, принадлежащей данному КОНТЕКСТу.

Контекст ВМ содержит все данные, до которых имеет доступ программа данного ВМ. Контекст включает и сам код этой программы. Это достаточно логично и, по крайней мере, с точки зрения реализации, необходимо хотя бы для обеспечения возможности считывания константной информации из кода или работы с самомодифицируемым кодом.

КС – это единственная величина, которая по логическим соображениям, а не только по соображениям оптимизации, должна храниться в ОУ.

Контексты различных ВМ могут по разному соотноситься друг с другом.

Они могут полностью не пересекаться. Например, контексты двух независимых программ. Один контекст может полностью охватывать или полностью принадлежать другому. Например, контексты двух процедур, статически вложенных друг в друга (контекст вложенной процедуры охватывает контекст статически охватываемой). Наконец, контексты могут пересекаться частично. Например, контексты двух процедур, статически вложенных в третью, более глобальную для обоих; или контексты двух функций, включающих общий глобальный контекст (глобальный контекст вложен в контексты обоих функций).

\\\\\(рис 1)///////*

Смена контекста

На одном и том же ОУ в различное время могут исполняться различные ВМ.

Во-первых, один ВМ может запускать другой, для получения результатов, необходимых ему согласно алгоритму. При этом на время работы второго запущенного ВМ, первый приостановит свою работу и возобновит её после завершения работы первого ВМ. Это обычный запуск процедуры.

* В настоящее время рисунки к данной статье находятся в работе и будут включены в статью в ближайшее время.

Во-вторых, работа текущего ВМ может быть прервана каким-либо внешним сигналом и ОУ перейдёт на выполнение системной процедуры. Затем система может переключить работу ОУ на продолжение вычислений одного из прерванных ВМ. Это прерывание и выход из него.

Каждое такое переключение работы ВМ сопровождается сменой КС в ОУ. Рассмотрим этот процесс более подробно.

При запуске какого-либо ВМ с самого начала, можно говорить о создании нового контекста и соответствующей КС с возможным помещением её в ОУ.

Этот новый контекст, вообще говоря, состоит из трёх частей:

- часть, передаваемая в новый контекст с помощью ссылки –
  это глобальные для нового ВМ данные (ГД)

- часть, передаваемая в новый контекст копированием значений – это параметры

- часть, создаваемая новым ВМ – это локальные данные.

Кроме этого следует считать, что в любой контекст входит стандартный для каждой ОС набор средств обращения к системе (вызовы системных функций и т.д.).

Отвлекаясь в данном разделе от деталей реализации, попытаемся определить общие методы и требования к процессу переключения контекстов.

Идентификация контекста

Если контекст А собирается переключиться на контекст В, то в контексте А должна, по крайней мере присутствовать идентификация контекста В. В общем случае следует рассматривать переключения непересекающихся контекстов. Учитывая этот случай, ясно, что идентификация контекста В не может быть представлена в виде обычной ссылки, т.к. в этом случае контекст В являлся бы частью контекста А.

Здесь возможны и на практике встречаются различные решения.

Может быть введён новый тип данных, содержащий ссылку на новый контекст, но не позволяющий работать с ним как со ссылкой. Его можно использовать только для переключения контекстов. Существуют и другие возможности. Здесь важно отметить, что должен существовать тип данных, идентифицирующий контекст.

Переключение контекста

Имея идентификацию контекста В контекст А не должен иметь возможность просто переключить контекст с А на В произвольным образом. Если контекст окажется переключённым, то старый код программы переключившей контекст уже, вообще говоря, будет недоступен для исполнения, так как он остался в старом контексте. Разрешить продолжить исполнение старого кода после переключения контекстов, кроме того, нельзя, так как это нарушило бы все принципы защищённости. Действительно, в этом случае «чужая» программа из контекста А стала бы модифицировать содержание контекста В.

Следовательно, вместе с переключением контекста, необходимо указать и номер исполняемой команды для продолжения счёта в новом контексте.

Переключение номера команды

Как и в случае идентификации контекста, контекст А не может, вообще говоря, содержать ссылку на код из контекста В, т.к. контексты могут быть и непересекающиеся.

В этом случае, как и в случае идентификации контекстов, необходим, вообще говоря, новый тип данных. Переменная этого типа может находиться в контексте А, указывая, на подобии ссылки, на код в контексте В. Но эту «ссылку» можно использовать только для переключения.

Реально, можно иметь один новый тип, включающий идентификацию, как контекста, так и номера команды для переключения. Тем более, как было выяснено ранее, оба эти переключения должны происходить одновременно. Будем называть этот тип МЕТКОЙ ВЫЧИСЛИТЕЛЬНОГО МОДУЛЯ (МВМ). Именно она может быть использована в качестве аргумента операции переключения контекста и кода программы.

Формирование переменных нового типа

Представляется довольно очевидным, что сформировать описанные выше переменные или МВМ можно только в контексте, где доступны ссылки на требуемый контекст и код, т.е. в контексте В. По существу это преобразование переменной типа ссылки в новый тип.

В результате последовательность действий выглядит следующим образом.

1. В контексте В (или в контексте, включающем в себя глобальные данные контекста В и код) создаются переменные нового типа необходимые для переключения в контекст В.

2. Эти переменные пересылаются или делаются доступными каким-либо другим способом в других контекстах (например, в А).

3. В нужный момент по программе контекста А происходит переключение в контекст В.

Обсуждение

Полезно представить другие рассуждения, приводящие к тому же результату.

Если контекст В вложен в контекст А, то все ссылки необходимые для переключения доступны в А и переключение из контекста А в В можно произвести без введения нового типа данных. Этот случай совсем не бесполезный и довольно часто используемый.

Например, в едином глобальном контексте можно запустить две процедуры (т.е. переключиться в новые контексты). В процессе работы каждый из контекстов пополнится различными локальными данными, контексты станут несовпадающими и защищёнными друг от друга.

Однако интерес представляет и случай переключения на не вложенный контекст.

Его можно представить как «делегированный вход». Т.е. подготавливается переход из контекста В на самого себя (или на подмножество В), но сделать это «поручается» другому контексту (А). Для этого подготовленные для переключения данные (ссылка на контекст В и код, доступные в контексте В) преобразовываются в новый тип в контексте В и передаются наружу в другой контекст (например в А), где и используются для переключения.

Здесь представлен наиболее общий случай с использованием МВМ. В реализации будут рассмотрены и более простые практические варианты этой схемы, которые являются её оптимизацией, но во всех этих схемах будут выдержаны следующие основные положения.

1. Точки входа в контекст формируются внутри самого этого контекста (ссылка на контекст и код).

2. Эта информация тем или иным способом делается доступной во внешних контекстах.

3. Переключение происходит одновременно (контекст и код).

Как уже указывалось, в случае запуска другого ВМ или в случае прерывания, текущий ВМ временно прерывает свою работу. Для того чтобы в последствии можно было бы его продолжить, система формирует вариант МВМ, который охватывает весь текущий контекст, включая параметры и локальные данные.

Эта МВМ должна быть где-то сохранена. Можно её передать в качестве параметра в вызванный ВМ. Запущенный ВМ в конце своей работы специальной операцией возврата возобновит работу прерванного ВМ. Однако этот вариант имеет проблемы.

Дело в том, что для обеспечения правильной динамической вложенности последовательности запусков ВМ, в конце работе запущенный ВМ должен вернуть управление прерванному модулю. Для этого МВМ, переданная как параметр, не должна быть забита или перемещена куда либо. Этого можно достичь, если для такой МВМ определить лишь операцию возврата в прерванный модуль и не вводить операцию считывания и пересылки. В таком случае МВМ не может быть забита другой МВМ. Если же она будет забита чем-либо другим, то во время возврата возникнет прерывание по контролю типов.

Другой, быть может более надёжный, способ заключается во введении отдельного массива, ссылка на который находится в ОУ. Этот массив не входит в пользовательский контекст, т.е. он не может быть считан или модифицирован программой, работающей в ОУ. Он доступен только аппаратным командам (находится в контексте аппаратуры). В этот массив аппаратура перепрятывает МВМы по стековой дисциплине (FIFO).

При запуске ВМ в ОУ появляется КС, которая в процессе счёта может перепрятываться. В конце работы ВМ команда возврата уничтожает эту КС. При этом система может уничтожить все локальные данные, которые в результате окажутся недоступными. Команда возврата, кроме того, может передать в вызвавший контекст данные, являющиеся результатом его работы.

Теперь можно уточнить понятие ВМ. Это отрезок работы системы с одним и тем же контекстом от момента создания его КС до её уничтожения. Однако иногда термин ВМ будет использоваться в статическом смысле, как программа работы в означенном контексте.

В системе должен быть предусмотрен важный механизм аварийных выходов из ВМ. Его обсуждение будет здесь опущено, чтобы не слишком усложнять описание, так как оно не нужно для понимания основных идей.

В качестве иллюстрации переключения, обсудим упоминавшееся ранее создание нового узла в ИП.

Ранее было более удобно представить эту функцию как элементарный примитив. Теперь можно сделать маленький шаг в сторону реализации и представить, как эта функция будет выполнена в операционной системе с помощью стандартных средств, введённых выше.

Напомним ещё раз, что каждый контекст имеет стандартную часть обращения к системе.

Для реализации функции создания нового узла, в упомянутый стандартный контекст включается МВМ модуля ОС реализующего эту функцию. Глобальный контекст этого модуля является частью контекста ОС, который не доступен вызывающим модулям. Для создания нового узла программа запускает этот модуль ОС с параметрами, описывающими размеры нового объекта. В процессе запуска происходит переключение контекста, и в ОУ выставляется глобальный контекст этого модуля, принадлежащий ОС. В этом контексте есть ссылки (или ссылка) на все ресурсы системы, в том числе на всю память (это реализация) и таблицы её занятости. Модуль выбирает соответствующую область, формирует на неё ссылку и возвращает её вызвавшему модулю. Таким образом, контекст пользовательского модуля пополняется новым узлом и в контексте этого пользователя появляется новая ссылка на него.

Анализ

В предыдущих разделах была изложена «идеальная» с точки зрения защищённости система. Ниже будет проведён её анализ и обсуждены её важнейшие свойства.

Защищённость

Как видно из изложенного выше, защищённость системы базируется на следующих простых основах.

(НИКТО НЕ ТРОНЕТ МЕНЯ)
Благодаря поддерживаемому системой механизму ссылки, только ВМ создавший некоторую область данных имеет до неё доступ. Никто другой не имеет до неё доступа, если ссылка на эту область не была добровольно передана её владельцем кому-либо другому.

(Я НЕ ТРОНУ НИКОГО)
С помощью механизма ссылки в любой данный момент времени формируется контекст работающей программы, строго ограничивающий множество доступных ей данных.

Рассмотрим, например, два ВМ с полностью разделёнными контекстами (глобальными и локальными).

Оба эти контекста полностью защищены друг от друга, в том смысле, что не могут обратиться к данным другого контекста, какие бы программы ни исполнялись в этих контекстах. Аналогично, каждый контекст защищён от влияния любых других, с которыми у него нет пересечения.

Однако эта защита существенно отличается от других подобных схем (UNIX или Windows), таких, например, как защита между двумя виртуальными пространствами, когда невозможно никакое взаимодействие.

В нашем случае эти два контекста могут активно взаимодействовать, работая в одной задаче.

Для этого достаточно, чтобы в глобальном контексте каждого ВМ имелась МВМ противоположного. В результате эти модули могут без ограничений вызывать друг друга, обмениваясь параметрами и возвращаемыми значениями. При этом во время работы одного модуля он не будет иметь доступа к данным другого.

Более того, если это необходимо по смыслу решаемой задачи, модули, через механизм передачи параметров и результатов, могут обменяться ссылками и работать над общими данными.

Результирующая защита предельно строгая, но она ни как не ограничивает возможности программиста.

В другом случае, когда контексты пересекаются, всё взаимодействие ограничивается этим пересечением. Это значит, что если модуль производит вычисления не обращаясь к общим данным, то он не испытывает ни какого влияния со стороны. В этом случае его работа полностью изолирована и ни как не зависит от работы других ВМ, даже если контексты пересекаются.

И только, если программа модуля обращается к общим данным, неправильное поведение партнёра может повлиять на правильность счёта. Однако все эти взаимодействия программа модуля сама планирует и известно строго ограниченное число мест, на которые следует обращать внимание. Программист всегда может правильно с его точки зрения выбрать степень взаимодействия, оценив соотношение риска и удобства плюс эффективность.

Это выгодно отличает данную систему от других, в которых правильность работы программы зависит от правильности или степени злонамеренности другой программы, без возможности своим поведением препятствовать отрицательному эффекту этих ошибок или злых намерений.

Теперь можно более точно определить, что в описываемой системе элементарным звеном защиты является контекст. Любой контекст защищён от неправильной или злонамеренной работы программ, работающих в других контекстах.

Иными словами, обеспечивается защита одной процедуры от другой. В Эльбрусе для обозначения этого типа защиты используется термин «контекстная защита».

Два заключительных замечания.

Можно обобщить понятие ссылки до более широкого понятия права, что-либо выполнять.

Это право должно быть представлено данными нового типа с соответствующими операциями. Это право может передаваться между контекстами, так же как ссылки или любые другие данные. И на его использование надо распространить приёмы контроля типов. В западной литературе для обозначения этого понятия используется термин **capability**.

О привилегированности.

Любую операцию можно себе представить, как вырожденный случай вычислительного модуля. Поэтому, если бы соображения эффективности позволяли это сделать, то каждая «привилегированная» операция могла бы быть представлена соответствующей МВМ. МВМ всех таких операций создавались бы аппаратурой во время инициализации системы и размещались бы в определённом контексте ОС. В процессе работы ОС могла бы размещать нужные МВМы в нужных контекстах. В этом случае не было бы необходимости вводить привилегированный режим, но тот контекст, в которых присутствуют МВМ «привилегированных» операций был бы эквивалентен привилегированному режиму.

Однако он более совершенен, чем обычный привилегированный режим, так как в отличие от привилегированного режима, который разрешает всё, данный режим разрешает только то, что необходимо в данном контексте. В принципе, можно себе представить несколько различных контекстов с различным набором «привилегированных» МВМ. Разрешается только то, что необходимо для работы данного контекста. Это повышает защищённость.

Эти соображения о привилегированности приведены только лишь для того, чтобы подчеркнуть фундаментальность использованных принципов. В Эльбрусе сохранён привилегированный режим, но, в отличие от других систем, только из соображений эффективности, так как использовать процедурный механизм для запуска простых операций не рационально.

Изложенные выше принципы построения защищённой системы вполне достаточны по существу и для предотвращения опасности вирусов. Однако без рассмотрения системы файлов и реализации этих принципов в ней, это утверждение не вполне понятно. Поэтому вопрос предотвращения угрозы вирусов будет более детально рассмотрен после обсуждения системы файлов.

Сокращение ошибок в программах

Изложенная система значительно облегчает обнаружение ошибок в программах.

Прежде всего, введение динамического контроля типов позволяет автоматически обнаружить целый ряд ошибок, которые без этого контроля приходилось бы обнаруживать сложным образом, через семантику программы.

Если, например, программа по ошибке попытается модифицировать арифметическими операциями ссылку, то в защищённой системе произойдёт диагностическое прерывание в месте ошибки, и ошибка может быть исправлена.

В традиционных системах такая ошибка приведёт к считыванию или модификации неправильной ячейки памяти без всякой сигнализации, и ошибка обычно проявляется значительно позже по неверным результатам, что значительно усложняет нахождения реального места ошибки.

Большое количество ошибок может быть обнаружено автоматически при введении типа данных «пусто» или неинициализированные данные. Это такой тип данных, которым система заполняет всё вновь выделяемое пространство. Прежде чем программа пожелает считать информацию из какого-либо места ИП, в него, в место находящегося там «пусто», должен быть присвоен результат какой-либо операции. Попытка использовать данные типа «пусто» в какой-либо операции вызывает диагностическое прерывание.

Один пример. Типичный класс ошибок Фортрана – описка в идентификаторе. При этом в Фортране автоматически и неявно заводится новая переменная. Наиболее вероятно первое обращение к ней будет по считыванию, просто потому, что считываний значительно больше, чем записей. С новым типом данных это вызовет диагностическое прерывание, и ошибка будет обнаружена автоматически.

Но, пожалуй, наиболее важным следствием введённого контроля является реальная поддержка модульного программирования.

Действительно, как уже отмечалось, любые ошибки внутри какого-либо модуля ни как не влияют на работу других модулей, так как во время работы модуля в ОУ хранится КС, которая ограничивает область доступа программ модуля. Какой бы ни была программа этого модуля (вплоть до случайного кода), система не даст возможности выйти за границы контекста. Любые ошибки в программе модуля проявятся только лишь в неправильных результатах выдаваемых этим модулем. Исключаются такие неприятные для отладки случаи, когда модуль выдаёт правильные результаты, но в качестве побочного эффекта портит информацию, принадлежащую другим модулям. Такую ошибку трудно локализовать. Надо быть готовым отлаживать уже отлаженные программы, данные которых были испорчены модулем, допустившим ошибку.

Другой, столь же неприятной, ошибкой является «дикая» передача управления. Локализация такой ошибки так же затруднена. Подобная ошибка, так же исключается в описываемой системе.

В результате, значительно облегчается отладка больших программных комплексов, так как значительно облегчается сборка всей программной системы.

Представим, что вычисления остановлены тотчас после завершения работы какого-либо модуля.

Сделать это не трудно, так как модуль завершится либо через возврат, либо через аварийный выход, это ограниченное число точек, в которых следует установить диагностические остановы – никаких других «диких» переходов за пределы модуля быть не может.

В этой точке можно проверить все результаты, вычисленные модулем и, если они правильны, то не следует ожидать неприятностей от работы этого модуля.

Ситуация несколько усложняется наличием глобальных переменных. Однако при современных методах программирования глобальные данные хорошо структурированы (объектно-ориентированные языки), так, что их количество, доступное в каждом модуле, сводится к необходимому минимуму и попытки выйти за его пределы контролируется контекстом.

Большой опыт эксплуатации систем Эльбрус показал значительное облегчение процесса отладки программ. По оценке большого числа пользователей время отладки сокращается на порядок. Значительно уменьшается число необнаруженных ошибок в программах переданных в эксплуатацию.

В качестве объективного подтверждения этого факта можно привести два примера.

На машины Эльбрус переносилось много **отлаженных** программ с других систем, написанных на языках. Во всех случаях такого переноса в этих отлаженных программах на Эльбрусе обнаруживались ошибки.

При прогоне в системе симуляторов Э3-Е2К широко распространённых тестовых программ SPEC92, которые до этого уже работали на всех известных в мире системах, были обнаружены около 30 ошибок.

По опыту разработки системных программ Эльбрус, описанная система имеет большой эффект при разработке операционной системы, так как при этом не только значительно облегчается отладка, но проще становится сама разработка, так как отпадает необходимость заботится о внутренней защите между модулями самой системы. По своему характеру системное программирование становится подобным прикладному. Алгоритмы ОС, вызванные из пользовательской программы или по прерыванию, работают на стеках пользователя и т.д. Иными словами, ОС становится набором описаний, глобальных для любой пользовательской программы.

Простота использования

Важной чертой описываемой системы является простота и естественность её использования.

Программисту нет необходимости делать что-либо специальное для обеспечения защищённости. Нет необходимости отдельно оговаривать права доступа и т.д.

Если по алгоритму необходимо что-то передать, то эта передача одновременно означает и передачу прав.

В практике других систем известны случаи, когда сложные действия в принципе могли бы улучшить защищённость, но их сложность приводит к тому, что программисты ими не пользуются. Логика здесь простая. Действия для обеспечения защищённости не ощущаются программистом, как абсолютно необходимые, так как совсем не очевидно будет ли нарушена защита именно в этом месте, а сложные действия без острой необходимости обычно забывают делать.

Описываемая система лишена таких недостатков.

Система Эльбрус имеет совершенную защищённость, и, несмотря на это, нигде в руководствах не содержится ни одного слова о защищённости, так как для этого ничего не надо делать специально, при этом защищённость будет полная.

Нет ограничений на запуск нескольких независимо откомпилированных программ внутри одной виртуальной памяти. Это облегчает общение таких программ между собой (передача параметров by reference и т.д.)

Упрощается само проектирование и реализация программ, что, безусловно, не может не сказаться положительно на их эффективности.

Простая доказуемость защищённости

Из небольшого числа изложенных выше простых принципов доказуемо вытекает полная и удобная в использовании защищённость. Это верно, по крайней мере, на уровне проектирования системы. Остаётся проследить, чтобы не вкралась ошибка в реализации этих принципов. Весьма ограниченное количество функций, описанных выше, ответственных за защищённость, значительно упрощает и повышает надёжность задачи реализации,

РЕАЛИЗАЦИЯ

Описанная выше организация вычислений обеспечивает высокий уровень защищённости. Нельзя сказать, что приведённые базовые идеи не были знакомы разработчикам современных систем. Однако все современные системы весьма далеки от этого уровня защищённости.

Ещё раз напомним причину такого положения дел, которая, по всей видимости, заключается в следующем.

Архитектурный облик сегодняшних систем создавался давно. В то далёкое время технологическая база была слаба и не допускала реализации идей защищённости без значительной потери эффективности.

Эффективность систем в то время имела значительно большее практическое значение, чем защищённость. Не было Интернета, мал был обмен программами и.д.

По соображениям совместимости, все эти старые решения фирмам приходилось поддерживать до настоящего времени, не допуская даже небольших нарушений совместимости, необходимых для внедрения защищённости.

Однако в последнее время ситуация изменилась радикально.

Машины стали очень быстрыми, что несколько снижает приоритет проблемы эффективности.

С другой стороны невероятно возросла роль защищённости в реализации информационных систем.

В результате существенно возрастает значение многолетнего опыта команды Эльбрус по созданию совершенных защищённых систем и решению проблемы совместимости.

Данная работа представляет результаты проекта Эльбрус в части защищённости. Результаты по совместимости будут изложены отдельно.

Последующие разделы будут посвящены описанию различных вопросов реализации, целью которой было сохранить все привлекательные качества описанной выше «идеальной» системы обеспечив, при этом, высокую эффективность.

Архитектура Эльбрус и языки программирования

Это не секрет, что языки и архитектура машин испытывают некоторое взаимное влияние.

При разработке языков их создатели заботятся об эффективной реализации на существующих машинах, а разработчики архитектур заботятся об эффективной реализации существующих языков.

Архитектура Эльбрус занимает по отношению к другим существующим архитектурам приблизительно такое же положение как язык высокого уровня к языку ассемблера.

Действительно, пожалуй, единственно приемлемым определением высокого уровня программирования может являться наличие типов данных с контролем правильности их обработки и основанной на этом защищённости.

Все другие признаки оказываются несостоятельными.

Возьмём, например, внешние синтаксические черты. В прошлом были разработаны такие языки, которые позволяют разобраться с этим вопросом. Например, PL/360 Вирта. Он был похож по синтаксису на язык высокого уровня, но без контроля типов. Его использование привело всех к единодушному мнению, что это вариант ассемблера.

Несостоятелен и такой признак как проблемная или машинная ориентация. Язык ЛИСП по общему признанию является языком высокого уровня, но для существовавших в прошлом ЛИСП машин, он был машинно-ориентированным языком.

Можно приводить и другие примеры.

Если произвести эксперимент и попросить программистов разложить языки на две кучки, то в кучку языков высокого уровня попадут только языки с типами и типовым контролем, а в кучку ассемблеров – языки без типового контроля.

Языки С и С++ окажутся посередине. В этих языках есть типы, но нет строго типового контроля.

Таким образом, введение защищённости в архитектуру Эльбруса это по существу поддержка высокого уровня программирования. И именно по этому здесь следует проявить осторожность – будет ли эта поддержка адекватна существующим языкам.

Поддержана должна быть сама основа программирования высокого уровня, а не конкретные черты конкретного языка.

Поддержка конкретного существующего языка, который, как уже отмечалось, в определённой степени ориентирован на традиционные архитектуры, означала бы косвенную ориентацию архитектуры Эльбрус на существующие машины, что было бы не желательно.

Практика разработки трёх поколений машин с архитектурой Эльбрус показала, что в этих условиях правильным подходом является следующий.

Архитектура поддерживает реализацию в аппаратуре базовых типов (с обязательным включением всех адресных типов), типовый контроль и формирование контекста с помощью ссылок.

На первом этапе (системы Эльбруса 1 и 2) были введены в аппаратуру ссылки (дескрипторы) на простые массивы и эффективная поддержка контекста статически вложенных процедур.

На втором этапе (системы Эльбрус 3М, Е2К) были введены в аппаратуру ссылки (объектные дескрипторы) на объекты объектно-ориентированного программирования, но контекст поддержан простой одноуровневый с глобальными данными.

Обе реализации достаточно универсальны и допускают реализацию любых языков. Каждая из них ориентирована на более эффективную реализацию более популярного в своё время стиля программирования.

На Эльбрусе 1,2 были реализованы все распространённые в то время (70е – 80е годы) языки, разрабатывать языковые системы было удобно, и качество реализации было высоким.

На втором этапе (Эльбрус 3М, Е2К) в соответствии с провозглашённым выше подходом, в аппаратуре были поддержаны механизмы объектно-ориентированного программирования как такового, используемого в широком спектре объектно-ориентированных языков. Однако,

кроме того, были поддержаны и отдельные черты языка C++, повышающие эффективность его реализации.

Представляется, что для существующей в настоящее время ситуации в программировании, такое решение оправдано, ввиду доминирующего положения этого языка в мире.

В целом, так же как и на первом этапе, выбранный подход обеспечил хорошую реализацию любых, в том числе объектно-ориентированных языков (например, JAVA).

Важный вопрос поддержки языков C и C++ требует обсуждения ещё одной проблемы.

Дело в том, что, как уже указывалось, эти языки в своей основе были разработаны, базируясь на общих принципах программирования высокого уровня, опираясь на введения типов данных. Однако при реализации, в угоду эффективности, в нескольких случаях, которые будут обсуждены позже, допускались отклонения от этих принципов. Например, разрешение присвоения целых в переменные типа указатель.

В целом это небольшое числа случаев приводит к тому, что программы, написанные на этих языках, по своей семантике нарушают защиту.

Эти программы можно исполнять на Эльбрусе в двоичных кодах x86. Для их исполнения в Эльбрусе 3М-Е2К, как уже указывалось, разработана специальная технология, которая будет описана в другом месте.

В Эльбрусе 3М-Е2К кроме защищённого режима и режима полной двоичной совместимости с x86, любую программу с языков можно откомпилировать и исполнить незащищённом режиме. Этот режим обеспечивает наибольшую производительность.

Перенос программ в исходных текстах на Эльбрус (в защищённый режим Эльбрус 3М-Е2К), как правило, позволяет обнаружить достаточно много ошибок в уже отлаженных программах.

Были тщательно изучены все допускаемые отклонения от строгой защищённости. Эти отклонения от защищённости вполне можно запретить, без заметной потери эффективности, так как частота их использования не велика. Все эти особенности будут более подробно обсуждены позже. Такой подход возвратил языкам C и C++ свойства защищённости.

Разумеется, при переносе необходимо исправить все ошибки (в том числе нарушающие защищённость) в программах переносимых на Эльбрус, что, само по себе является положительным результатом, без которого не возможно обеспечение защищённости.

Важным свойством подхода Эльбрус является обеспечение обратного перехода. Программы, отлаженные на Эльбрусе после перетрансляции, должны работать на других системах. Эти программы будут значительно лучше отлажены, и не будут представлять угрозы для других даже на существующих машинах. Таким образом, Эльбрус может стать системой, на которой будет необходимо проверять любые программные системы перед выпуском их на рынок или перед передачей их пользователям.

Проблемы традиционных систем

В реальных машинах информационное пространство размещается в памяти (физической, виртуальной, в файлах).

В традиционных системах в качестве ссылки используется машинное слово, которое не сопровождается указанием типа (ссылка) и представляет адрес в памяти данных, на которые указывает ссылка.

Такая реализация страдает, как минимум, двумя недостатками.

Во-первых, из соображений эффективности, такая ссылка указывает лишь на начало области данных адресуемых этой ссылкой. Она не включает в себя размера области и прав доступа. Это лишает возможности системе во время работы провести контроль правильности выполнения практически любой операции со ссылкой. Как правило, это нельзя проверить и во время трансляции. Неверная или злонамеренная программа может легко выйти за границы адресуемой области.

Во-вторых, такая ссылка не может быть надёжно идентифицирована системой во время работы, так как не содержит признака ссылки. Таким образом, ошибочная или злонамеренная программа может произвольно изменять эту «ссылку» или, ещё того хуже, использовать произвольные данные из памяти в качестве ссылки.

Введение различных «ключей», «рингов» и т.д. усложняет систему, но совершенно не решает проблему.

В таких условиях нет никакой возможности ограничить работу текущего модуля, каким бы то ни было контекстом.

Такая система, по крайней мере, на ассемблерном уровне лишена всякой защиты.

Системы программирования на языках высокого уровня пытаются восполнить этот недостаток. Современные языки пытаются во время трансляции выполнять контроль типов, который необходим, чтобы ссылка вернула себе свойства, представленные в описанной выше «идеальной» системе. Этот контроль должен гарантировать, что истинные ссылки не будут модифицированы произвольными операциями и, что произвольные данные не будут использованы в качестве ссылки.

Необходимо ввести ограничения на язык, чтобы такой контроль был бы возможен, хотя бы в части случаев. Таким ограничением является статика типов. Т.е. тип данных (в том числе информация о том, является ли переменная ссылкой и какой именно) связывается с переменной во время её объявления на всё время её существования, что даёт возможность компилятору выполнить необходимый контроль во время трансляции программы.

Существуют языки, очень последовательно выполняющие этот подход – например Алгол 68.

Существуют и более радикальные случаи, например Java, в которой явные ссылки на произвольные данные вообще исключены из языка. Оставлены лишь ссылки на объекты.

Однако практика показывает, что чем последовательнее проводится этот принцип, тем менее универсальным становится язык, тем труднее на нём программировать сложные задачи, такие, например, как системные программы, тем, в конечном счёте, он становится менее эффективным на этих задачах.

Трудности статических языков проявляются, например, когда динамически вычисляется индекс переменной в массиве и нужно проверить, не выходит ли он за границу этого массива.

Другой проблемой является переменная, в которой желательно хранить данные разных типов. В некоторых языках в переменную со статически определённым типом можно производить присвоение данных некоторых других типов с автоматическим преобразованием типа к типу переменной, в которую происходит присвоение. Но проблемой является случаи, когда преобразования невозможны или не желательны, и в переменной необходимо хранить данные различных типов.

Наконец, непреодолимые трудности возникают, когда одна независимо оттранслированная программа вынуждена работать с данными из другой независимо оттранслированной программы. Примером может служить ОС, работающая с данными независимо оттранслированных программ пользователя. Другой пример - это обмен данными (включая ссылки) с независимо оттранслированной программой, считанной из сети INTERNET.

Практика ограничений оказалась не очень успешной.

Алгол 68 оказался не очень практичным. Java не эффективна и не универсальна. Доказательством является хотя бы тот факт, что сама система Java написана на C.

Распространение получил язык C, в котором, в угоду эффективности, победившей защищённость, работа со ссылками просто ни как не контролируется.

Справедливости ради следует отметить, что практика использования этого языка проявила отрицательное отношение к его чертам, нарушающим защищённость. Это выражается в том, что в важнейших случаях, таких, например, как присвоение целого в переменную типа указатель, компиляторы выдают предупреждения.

Подход Эльбруса

Подход Эльбруса заключается в том, что ссылки не выбрасываются из языка подобно языку Java, но, напротив, работа с ними делается эффективной и не требующей статического контроля благодаря аппаратной поддержке. В этом случае нет необходимости как-либо ограничивать язык, и даже ассемблер такой машины не нарушает защищённости, подобно самым строгим языкам высокого уровня.

В дальнейших разделах приводятся основные трудности по обеспечению эффективной реализации системы, встретившиеся при проектировании Эльбруса, и обсуждаются принятые решения.

Все проблемы можно условно разбить на две группы: реализация ссылки и контекста.

Ссылка

Для того чтобы ссылка сохранила все черты, приведённые в описании «идеальной» системы, без расчета на статический контроль в процессе компиляции, необходимо:

- чтобы все данные в памяти содержали информацию об их типе, позволяющую отличить ссылку от прочих данных. Эта информация позволит аппаратным операциям вести контроль своих аргументов и исключить использование обычных данных в операциях со ссылками и ссылок в других операциях

- чтобы все работа со ссылками выполнялись аппаратными операциями для гарантии их правильности

- чтобы ссылка содержала достаточно информации для правильной работы операций.

Тип данных в памяти (ТЕГи)

Одним из наиболее существенных отличий описываемой системы от традиционных является введение **дополнительных** разрядов в памяти, которые описывают тип информации, хранящейся в каждом слове.

К сожалению, это должны быть дополнительные разряды, т.к. все информационные разряды доступны для обработки обычными операциями, и их число нельзя сокращать, чтобы выделить разряды под тип.

ТЕГ – число битов

Число необходимых дополнительных битов (ТЕГ) зависит от выбранного подхода. В Эльбрусе 1, 2 каждое 64 разрядное слово сопровождалось дополнительными 8-ю битами, в которых кодировался тип данных и контрольная информация для обнаружения сбоев. С помощью этих разрядов кроме различных вариантов ссылок кодировались и другие типы данных, такие как вещественные числа и др.

Для защиты данных от внешнего воздействия, защиты памяти достаточно специально кодировать только ссылки и быть может ещё несколько специальных типов данных, будем их обозначать СС. Более детальная кодировка других типов помогает в диагностике ошибок внутри модуля, что так же часто оказывается полезным.

Однако всё же наиболее значимым является защита памяти и межмодульная защита. Кроме того, как оказалось, специальная кодировка общераспространённых типов (целые, вещественные, логические биты и т.д.) не очень совместима со сложившейся практикой программирования в современных языках. Такой чрезмерный контроль типов, улучшая

диагностику, накладывает некоторые ограничения на стиль программирования, несколько отличный от стиля, сложившегося в настоящее время.

Это никак не сказывается на возможности реализации языков, однако при несоответствующем стиле, система программирования вынуждена генерировать дополнительные преобразования типов, что сказывается на эффективности.

При разработке Эльбруса 1 в середине 70-х в России не было ещё языка С и сложившегося стиля программирования на нём, что и продиктовало выбранное решение в сторону лучшей диагностики. Кроме того, в то время память машины изготавливалась индивидуально для Эльбруса, и реализация 8ми дополнительных разрядов на слово не представляло проблемы.

При проектировании Э3-Е2К, в отличие от Эльбрусов 1, 2, было введено только по 2 дополнительных бита на каждое 32-х разрядное слово, с помощью которых можно отличить СС от всех других общеизвестных типов данных, которые данными средствами не различимы между собой. Для более детальной кодировки СС и различия между различными СС, используются информационные разряды.

Причина введения такого числа дополнительных разрядов заключается в следующем. Дело в том, что СС может быть различного размера (32, 64, 128 бита).

Если все эти данные кодировать одним битом, то невозможно будет отличить, например, два СС по 32 бита от одного в 64 разряда.

Кроме того, можно было бы, например, записать в одну из половинок 64-разрядого СС в памяти, СС в 32 бита, что эквивалентно выполнению операции, не предусмотренной для данного типа данных.

Чтобы избежать этих неприятностей, принимаются следующие меры.

1. Два бита тегов используются для кодировки следующих четырёх случаев использования 32-х битового слова:

   - обычная информация (не ссылка, не СС)
   - 32-х битовое СС
   - часть 64-х битового СС
   - часть 128 битового СС

2. Все типы СС располагаются в памяти когерентно (младшие разряды адреса СС в 32, 64 и 128 бита имеют 2, 3 и 4 нулевых бита соответственно)

3. Все аппаратные операции (в том числе LOAD и STORE) работают только с целым СС, нет операций работающих с частью СС.

Легко убедиться, что приведённые меры исключают возможность возникновения описанных выше неприятностей.

Подобная же проблема существовала в Эльбрусе 1, 2 и решена она была подобным же образом.

Принятое решение допускает и более сложные случаи, один из которых был реально использован.

Например, конкретный тип CC128 может иметь три старших 32-х разрядных слова, закодированных как CC128, а младшее слово как обычная информация. Все операции, работающие с данным типом, опираясь на код типа внутри CC, будет контролировать соответствие такой кодировки. Это позволяет для данного конкретного типа CC выполнять соответствующие операции как с единой 128 разрядной величиной и в то же время работать с младшим 32-х разрядным словом независимо, как с обычными 32 разрядными данными. Разумеется, это должно соответствовать семантике данного типа. Например, сама 128 разрядная величина может быть ссылкой описывающей массив, а последние 32 разряда могут представлять текущий индекс в этом массиве.

ТЕГ – хранение в памяти

В Э3-E2K используются стандартные кристаллы памяти и/или SIMM, DIMM, RIMM.

Для хранения тегов используются ECC биты (биты контроля и коррекции аппаратных ошибок). При этом мощность корректирующих возможностей не сокращается, просто увеличивается объём битов, охватываемых одним ECC кодом, и освободившиеся в результате кодовые комбинации используются для хранения тегов.

ТЕГ – хранение на файлах

Обсуждаемые ссылки адресуют оперативную память. Это значит, что они не нужны в системе файлов. Никакая система файлов не требует или не рассчитывает на использование ссылок в память (ссылки внутри системы файлов будут рассмотрены в разделе ФАЙЛЫ).

Таким образом, в пользовательских файлах не следует реализовывать ссылки в память, а, следовательно, не нужно вводить теги.

Однако, к сожалению, существует, не большое по объёму данных исключение. Это файл, хранящий виртуальные страницы памяти, откаченные на диск. Ссылки, хранящиеся на этих страницах должны откачиваться в файл и подкачиваться обратно в память.

Существуют два способа для реализации такой возможности.

Как откачку, так и подкачку можно производить с использованием промежуточного буфера в памяти. При откачке, откачиваемую страницу необходимо программно переслать в промежуточный буфер, объёмом, несколько превышающим страницу. При этой пересылке теги всех слов откачиваемой страницы помещаются в дополнительные слова буфера. В результате, сам буфер не использует свои разряды тегов. Затем буфер пересылается на диск.

При подкачке всё происходит в обратном порядке.

Второй способ предусматривает прямую откачку/подкачку без использования промежуточного буфера, но в этом случае упаковку тегов должна выполнять аппаратура ввода/вывода.

В Эльбрусе 1, 2 были использованы оба метода. При откачке страницы на барабан (в Эльбрусе было такое устройство), упаковку производила аппаратура, при откачке на диск теги паковала программа ОС с использованием промежуточного буфера.

В Э3-Е2К так же могут использоваться оба метода, в зависимости от того, есть ли в чип сете данной модели специальный контроллер упаковки.

Это решение не нарушает защищённости, так как пользовательские программы не имеют непосредственного доступа к откаченным на диск данным.

Структура ссылки

В принципе ссылка имеет простую структуру. В ней содержится, в основном, описание области, на которую ссылка указывает и права доступа, например, признак разрешения записи в эту область через данную ссылку (признак константы).

Однако описание области может сильно варьироваться в разных случаях.

В простом случае ссылки в Эльбрусе 1, 2 (там она называлась ДЕСКРИПТОРом) описание состоит из базового виртуального адреса описываемой области и её размера. Для считывания или записи какого-либо слова из описываемой области, как и в традиционных машинах, в операции LOAD\STORE подаются два аргумента: база и смещение. Однако, в отличие от традиционных машин, база представляет не просто целое, а описанный выше ДЕСКРИПТОР.

В традиционных машинах операция обращения в память для получения исполнительного адреса производит сложение двух аргументов. В случае же Эльбруса – это не просто сложение, а операция индексации. Исполнительный адрес получается так же, путём сложения базового адреса из дескриптора со вторым аргументом – индексом. Однако, кроме этого, индекс сравнивается с размером области из ДЕСКРИПТОРа и в случае выхода за границы области, происходит диагностическое прерывание. Индекс и размер области могут быть выражены в битах, байтах, словах, двойных словах и квадро. В ДЕСКРИПТОРЕ содержится указание об этом.

ДЕСКРИПТОР Эльбруса 1, 2 занимает 64 бита.

В Э3-Е2К ссылка на данные имеет несколько другую структуру, так как в этих машинах поддерживаются объектно-ориентированные языки. ДЕСКРИПТОР ОБЪЕКТА (ОД) содержит базовый адрес объекта и несколько малоразрядных полей, описывающих публично доступную и приватную области объекта. Каждая область описывается двумя полями: смещением относительно базового адреса и размером. Кроме этого ОД содержит поле, описывающее тип объекта, на который ссылается данный ОД.

Доступ в публичную область происходит подобно случаю ДЕСКРИПТРа Эльбруса 1, 2, с той лишь разницей, что для вычисления исполнительного адреса теперь необходимо сложить три

числа: базовый адрес и смещение публичной области из ОД и индекс – второй аргумент операции обращения в память (разумеется, производится контроль выхода из публичной области). ОД занимает 128 битов.

Это, разумеется, не единственный возможный способ представления ОД. Можно было бы использовать вариант более простой и близкий к Эльбрус 1, 2, но в случае разрешённых обращений к приватным данным объекта, пришлось бы использовать операцию преобразования дескриптора (CAST см. далее).

Ещё один формат ДЕСКРИПТОРа размером в 128 бит, введён для реализации указателя на массив. К сожалению, в этом месте для эффективности реализации языков С и С++ пришлось аппаратно поддержать ошибочную, с нашей точки зрения, черту этих языков, когда не различаются указатели смотрящие на элемент массива или на весь массив. Для сочетания обоих этих свойств, пришлось в формат ДЕСКРИПТОРа включить базовый адрес массива, его размер, а так же, в виде целого, индекс элемента массива, на который в данный момент смотрит указатель.

Как уже указывалось в разделе *ТЕГ – число битов*, этот формат выполнен следующим образом. Три 32 битовых его слова имеют теги СС128, а последнее 32 битовое слово имеет тег обыкновенного целого. Специальные аппаратные операции, введённые для работы с этим ДЕСКРИПТОРом, работают со всеми 128 битами как с единым ДЕСКРИПТОРом. Пересылка его, например, происходит одной операцией. Они всегда проверяют правильность всей структуры ДЕСКРИПТОРа. Но с 32 битовым целым, входящим в его состав, могут работать обычные операции.

В результате, к указателю можно добавлять или вычитать произвольные значения. Эти операции будут выполняться с целым индексом. А при операциях обращения в память аппаратура проверяет выход индекса за границы массива.

Представление ссылки на функцию будет рассмотрено ниже, после обсуждения реализации контекста.

Аппаратные операции со ссылками

В отличие от традиционных систем, в этой машине все операции LOAD/STORE в качестве аргумента представляющего адрес памяти используют не простое число, а ссылку.

Кроме этого вводятся специальные операции адресных преобразований. Типичным примером является операция индексации, которая, получив в качестве аргументов ссылку на массив (дескриптор массива) и индекс, вырабатывает в качестве результата ссылку на элемент этого массива, соответствующий указанному индексу.

Другим примером важной операции, существующей во всех объектно-ориентированных языках, является операция преобразования типа ОД к базовому типу (CAST). По своему существу это операция адресного преобразования, которая должна скорректировать малоразрядные поля в ОД, описывающие публичную и приватную части объекта.

Чтобы эти преобразования не противоречили защищённости, приняты следующие меры.

Сама операция преобразования производится с помощью специальной аппаратной операции (CAST), аргументами которой являются преобразуемый ОД и специальная тегированная константа. Ввиду того, что константа тегирована, она не может быть создана в пользовательской программе. Как ОД, так и константа содержат тип объекта, и операция проверяет их соответствие между собой, что и гарантирует защищённость.

Создаётся эта константа специальной системной программой, которую можно считать частью аппаратуры, во время генерации типа объекта (класса) и помещается в статическую память типа объекта (класса).

Эта же программа создаёт «болванку» для генерации нового объекта данного типа. Эта «болванка» практически совпадает с ОД данного типа, только не содержит виртуального адреса объекта. Программа, имеющая такую «болванку» в своём контексте, может передать её системной процедуре в качестве параметра, в результате эта системная процедура создаст объект и выдаст пользовательской программе ОД (оператор «new»).

Работает эта системная программа по информации, предоставляемой транслятором, но она имеет полную возможность проверить эту информацию так, чтобы она не противоречила правильной семантике объекта, а это и есть обеспечение защищённости.

Операция CAST достаточно часто неявно используется в программах, при многих пересылках ОД. Поэтому считывание константы при каждом исполнении этой операции может заметно понизить эффективность программы в целом. Для повышения эффективности, предусмотрена оптимизация – размещение константы литерально в командном потоке, рядом с самой операцией CAST. Такое решение, безусловно, повышает эффективность, так как константа фактически присутствует в команде литерально, но требует специальных мер для сохранения защищённости, так как теперь эту константу делает пользовательская программа. Здесь следует застраховаться от ошибок или преднамеренных атак со стороны компилятора. Та же проблема возникает для ассемблерных программ.

Спасти дело может довольно простая и быстро работающая системная программа верификации кода, подлежащего исполнению. Она должна проверить каждую литеральную константу и убедиться, что такая же константа существует в статической памяти класса. После выполнения верификации в документацию файла сама системная программа верификации ставит признак выполненной верификации. Любая запись в данный файл кода стирает этот признак.

Таким образом, исключается возможность выполнения не верифицированных программ а, следовательно, гарантируется правильность констант и сохранение защищённости.

Для того чтобы программа верификации могла проверить произвольную программу, система команд должна быть спроектирована таким образом, чтобы код можно было бы синтаксически разобрать целиком. Системы команд современных машин не обладают этим свойством. В Э3-Е2К оно включено. Этот вопрос будет обсуждён позже.

Литеральная константа CAST - это единственная причина, где нужна программа верификации.

В принципе можно оттранслировать программу без данной оптимизации и пользоваться константами из памяти класса. В этом случае верификация не нужна. Поэтому верификация проводится по сигналу прерывания, когда исполняемая программа собирается использовать литеральную константу и отсутствует признак проведённой верификации.

Компактировка

При работе со ссылками необходимо обсудить следующую проблему.

Ссылка содержит виртуальный адрес, который генерирует система в момент создания объекта или массива, на который указывает ссылка.

Если массив или объект, на который указывает ссылка, уничтожается в процессе решения задачи, то этот отрезок виртуальной памяти освобождается.

Однако его нельзя использовать вновь для размещения нового объекта, так как в программе могут остаться зависшие ссылки, при обращении через которые защищённость будет нарушена.

В связи с этим система удовлетворяет запросы пользовательских программ на виртуальную память последовательно, двигаясь в сторону старших адресов и не переиспользуя освободившиеся виртуальные адреса.

Когда объём свободной виртуальной памяти приближается к исчерпанию, включается алгоритм компактировки.

Этот алгоритм производит последовательный однократный просмотр всей оперативной памяти задачи и выполняет следующие изменения:

- уничтожает ссылки на уничтоженную виртуальную память,

- компактирует память, сдвигая массивы в сторону младших адресов в плотную последовательность и корректируя соответствующим образом адреса ссылок.

В результате вся занятая виртуальная память не будет иметь «дыр», а вся свободная память оказывается в виде одной непрерывной области.

Практика показывает, что этот алгоритм вызывается крайне редко и, в результате, не оказывает заметного влияния на эффективность решаемой задачи.

Поддержка программно определяемых типов данных

Как видно из вышеизложенного, основой философии Эльбруса является аппаратная поддержка и строгий контроль правильности работы с типами данных.

Основная идея
реализации типового контроля заключается в следующем.

Система должна обеспечить возможность произвольного распространения данных определённого типа пользовательскими программами, разумеется, в соответствии с принципами, изложенными выше (ограничения контекста, доступ через ссылки и т.д.). Т.е. эти данные должны быть расположены в контексте пользовательских программ

Но вся работа с внутренним содержимым переменных этого типа (включая их генерацию) должна быть сосредоточена в определённом программном или аппаратном модуле (в отдельном контексте) вообще говоря, отличном от того контекста, где располагается объект в целом. Это свойство должно быть обеспечено независимо от ошибок в программах, использующих эти переменные.

Для надёжного обеспечения выполнения этих требований без введения ограничений на языки необходима явная идентификация типа переменных в памяти.

Явная идентификация типа переменной необходима для того, чтобы алгоритм обработки мог провести контроль правильности типов данных, поданных ему на вход и отсеять возможные ошибки в программах пользователя.

Аппаратные типы данных

В Эльбрусе 1, 2 архитектурно поддержаны базовые типы, для которых существуют аппаратные операции. В Эльбрусе 3М-Е2К аппаратно поддержаны все типы данных, содержащие адреса (указатели и т.д.) и «пусто».

Переменные этих базовых типов укладываются в 32-128 битов. Вся работа с внутренним содержимым переменных обеспечивается аппаратными операциями (что эквивалентно отдельному программному модулю), которые и производят контроль типов на своём входе.

Разумеется, данные могут присутствовать в смешанных массивах, поэтому явная идентификация их типа должна сопровождать каждую такую переменную и не может быть вынесена в ссылку на массив. Использовать ссылки для каждой переменной расположенной в массиве совершенно не практично.

Как уже это описано выше, проблема аппаратных типов решается введением тегов и типового контроля в аппаратных операциях обработки.

Программно определяемые типы данных

В Эльбрусе 3М-Е2К, кроме аппаратных, архитектурно поддержаны и программно определяемые типы данных. По существу, в значительной степени, поддержка программно определяемых типов данных – это и есть поддержка объектно-ориентированного программирования. Класс – это и есть программно определяемый тип данных, а объект – это конкретный экземпляр данных этого типа.

Одним из важных отличий от аппаратных типов является тот факт, что эти объекты, как правило, имеют большие размеры, так, что обращение к ним всегда ведётся через ссылку и это допустимо с точки зрения эффективности.

Ввиду того, что обращение происходит через ссылку, тип такого объекта можно хранить в ссылке, т.е. в ОД.

Любая программа, которая имеет ОД в своём контексте, свободно может обращаться к публичным данным объекта. Для этого в командах обращения в память есть специальный признак, говорящий о том, что обращение идёт к публичным данным, и в этом случае не требуется ни какого дополнительного контроля, кроме контроля выхода за размер дескриптора.

Однако согласно *основной идее,* изложенной выше, доступ к приватным данным должен быть разрешён только из программ обработки данного типа. Обеспечивается это следующим образом.

Если в командах обращения в память по ОД стоит признак приватных данных, то для разрешения обращения проверяется специальный регистр в процессоре, который хранит тип объекта, когда работают программы обработки данного типа. Таким образом, внутри программы обработки данного типа становятся доступными приватные данные объектов этого типа.

Вопрос установки этого регистра будет рассмотрен позже.

Контекст

Реально контекст модуля состоит из данных хранящихся в оперативной памяти и в файлах. Контекст файлов будет рассмотрен позже, здесь же мы обсудим контекст в оперативной памяти.

Как уже отмечалось, контекст работающего модуля призван обеспечить доступ к информации необходимой для работы этого модуля и в то же время ограничить его работу этим объёмом.

В принципе это можно сделать с помощью единственной контекстной ссылки (КС), расположенной в процессоре. Однако процессор во время своей работы должен был бы очень часто считывать данные через КС, поэтому здесь уместна оптимизация. Классическая оптимизация заключается в следующем.

Дело в том, что контекст, описываемый с помощью КС, обычно состоит из нескольких непрерывных областей (например, данные глобальные, локальные, локальные данные различного уровня, различные общие данные ФОРТРАНа и т.д.). Для каждой такой области можно предусмотреть в процессоре отдельный регистр базы. Это значительно повысит эффективность, но требует предусмотреть механизмы загрузки и перезагрузки этих регистров.

Эта часть сильно зависит от организации системы команд и по-разному реализована в Эльбрус 1, 2 и в Эльбрус 3М-Е2К.

Эльбрус 1, 2 поддерживает статическую вложенность процедур, поэтому для эффективной реализации доступа к локальным данным различного уровня вложенности в процессоре предусмотрен набор ДИСПЛЕЙ регистров, которые хранят ссылки на локальные данные всех процедур, статически охватывающих работающую.

Эти регистры правильным образом корректируются аппаратным процедурным механизмом. Можно было бы и не вводить этих регистров, а хранить все ссылки на статически охватывающие данные в локальных данных каждой процедуры или хранить ссылку на одну охватывающую процедуру. ДИСПЛЕЙ регистры введены для оптимизации.

В Эльбрус 3М-Е2К контекст реализован иначе.

Ввиду изменившегося стиля программирования, преобладать стало одноуровневое программирование с общими глобальными данными (частный случай предыдущей схемы). В соответствии с этим в процессоре был реализован лишь два «дисплея» - ссылки на глобальные и локальные данные. Глобальные данные располагаются в памяти, а локальные в памяти и в регистровом файле процессора.

Подобный набор базовых регистров для доступа к данным из программы использовался и в традиционных архитектурах. Существенная деталь необходимая в реализации Эльбрус заключается в том, что каждый базовый регистр – это не просто адрес, но полноправная ссылка. На этих регистрах, кроме адреса, хранится и размер доступной области. Аппаратура проверяет и отсеивает попытки выхода за её пределы. Это касается не только адресов памяти, но и адресов в регистровом файле.

Переключение контекста

Переключение контекста – это переход процессора с работы по программе одного модуля на другой. По существу это происходит при запуске процедуры или возврате из неё.

Таким образом, становится ясно, что в защищённой системе с использованием типов, аппаратная поддержка механизма процедур становится обязательной, т.к. именно данные, принадлежащие процедуре, являются объектом защиты.

Как уже упоминалось, реальный создаваемый полный контекст (ПК) вычислительного модуля состоит из нескольких частей. Для нужд текущего описания объединим их в две группы.

Первая группа – это глобальный контекст (ГК), куда входят системная часть, контекст файлов и оперативный глобальный контекст. Это та часть контекста модуля, которая уже существует вне создаваемого контекста в момент его создания, как часть другого контекста и подаётся во вновь создаваемый контекст с помощью ссылки.

Вторая группа – это оперативный контекст (ОК), куда входят параметры и локальные данные. Эта часть создаётся во время создания самого полного контекста и запуска вычислительного модуля.

В случае запуска процедуры, полный контекст, с которого процессор переключается, сохраняется. А новый контекст, на который происходит переключение, создаётся.

В случае возврата из процедуры, оперативный контекст процедуры, с которой происходит переключение, прекращает своё существование. И процессор переключается на другой сохранённый полный контекст.

Это обычный процесс запуска и возврата из процедуры. В реализации Эльбруса есть некоторые особенности, на которых здесь не имеет смысла останавливаться подробно. Важно лишь отметить, что вся работа аппаратуры при этом происходит в строгом соответствии с описанными выше принципами.

Обсудим теперь подробнее процесс переключения, с точки зрения доступа к информации.

Изменение доступа происходит по двум, хотя и связанным между собой, но всё-таки разным, причинам. Хотя оба этих механизма уже обсуждались, полезно привести их здесь ещё раз.

Первый механизм – это само переключение контекста, которое приводит к изменению информации, доступной через новую КС.

Второй механизм – это доступ к внутреннему содержанию переменной какого-либо типа, если в новом контексте есть право такого доступа.

В Эльбрусе 3М-Е2К – это доступ к приватным данным объекта какого-либо типа. Он разрешён, если на спецрегистре процессора записано тегированное значение представляющее этот тип.

Можно сказать, что по такому же механизму работает аппаратура, когда она выполняет операции со ссылками. Можно считать, что в контексте аппаратуры есть этот тип (ссылка) и поэтому, когда включается этот контекст (работает аппаратная операция) она может читать и менять содержимое ссылки.

Эти механизмы связаны, так как, с одной стороны, доступ к тегированному типу управляется контекстом. С другой стороны, сам контекст работает благодаря правильному функционированию механизма ссылки, основанному на типовом доступе.

Но эти механизмы разные. Так как, например, если некоторая область памяти входит в два разных (но, разумеется, пересекающихся) контекста и эти контексты имеют различные права доступа к типам (различный набор тегированных переменных, представляющих тип), то приватные данные объектов различного типа будут по-разному доступны из разных контекстов.

Рассмотрим теперь более подробно сам процесс переключения контекста (или процедурные переходы).

Требования к процессу переключения

Как уже отмечалось, при переключении должно выполняться два требования обеспечивающих сохранение модульности и защищённости в системе.

Во-первых, при переключении с контекста А на контекст В изменение номера команды (сам переход) и смена контекста должны происходить одновременно, с точки зрения программ, участвующих в переключении, как единый и неделимый шаг.

Иными словами, нельзя переключить контекст с А на В, затем продолжить выполнения некоторых программ из А и лишь после этого перейти на программы В. Или наоборот, перейти на программы В, поработав некоторое время, переключить контекст,

Во-вторых, в общем случае значения контекстной ссылки и метки программы В, на которые происходит переключение, должны формироваться в контексте самой программы В. Затем, эта информация в виде тегированной величины (т.е. в виде значения защищённого от посторонних изменений), передаётся одним из многих возможных способов в контекст А и используется для переключения.

С точки зрения программирования – это вполне естественно. Тегированная величина формируется там, где доступны по контексту исходные данные, в неё входящие.

В качестве иллюстрации можно привести запуск формальной процедуры (с соответствующим переключением контекста) в языках со статической вложенностью.

Метка формальной процедуры (тегированная величина, содержащая контекстную ссылку и метку кода) формируется в статически охватывающей процедуре, где доступны глобальный контекст и метка входа будущей процедуры. Эта метка, затем, передаётся «наружу», в точку вызова формальной процедуры и используется для её запуска.

Разумеется, на практике в ряде случаев используются оптимизации, повышающие эффективность, но полностью эквивалентные описанной схеме.

С точки зрения возможной эффективности реализации, следует различать два случая.

Первый случай

Первый случай, когда глобальный контекст вновь запускаемой процедуры либо равен, либо даже является подмножеством полного контекста процедуры, с которой происходит переключение.

Примеры.

В языках С, С++.
Запуск одной глобальной процедуры из другой.

В языках со статической вложенностью.
Запуск статически вложенной процедуры из статически охватывающей.

Второй случай

Второй случай, когда глобальный контекст запускаемой процедуры включает в себя части, не входящие в полный контекст запускающей.

Примеры.

В языках C, C++.
Запуск процедуры (метода), входящей в объект, из глобальной процедуры или метода другого объекта.

В языках со статической вложенностью.
Запуск формальной процедуры.

Реализация первого случая

Первый случай допускает значительные оптимизации. Т.к. в точке запуска доступен глобальный контекст запускаемой процедуры, нет необходимости создавать формальную метку и тут же её использовать. Новый глобальный контекст и метку перехода можно литерально указать в команде перехода. (Обсуждение возможности указывать номер команды литерально см. раздел *Единица компиляции*). Следует только выполнить второе условие переключения – его неделимость. Это выполняется в рамках одной команды перехода.

В результате команда запуска процедуры в таком случае подобна обычному переходу существующих машин, в ней лишь добавлено указание о новом контексте. Этот новый контекст, вообще говоря, является подмножеством старого и, следовательно, для правильной защищённости должно быть выполнено ограничение контекста для новой процедуры. Так как ссылки на контекст размещены на спецрегистрах, то это указание сводится к статической информации о тех регистрах, которые должны быть доступны после переключения.

В Эльбрусе 1, 2, для языков со статической вложенностью – это отрезок регистров 0-N, где N лексикографический уровень запускаемой процедуры. В команде присутствует только это число N. \\\\\(рис.)/////

Случай C, C++ и того проще, так как глобальный контекст не меняется и в команде не нужна ни какая информация о контексте. Нужен лишь адрес перехода.

Реализация второго случая

Второй случай в принципе более сложен. Для языков с блочной структурой предусмотрена команда формирования формальной метки. Она включает в себя ссылку на контекст и на код. По существу, это самый общий случай переключения.

Команда перехода на формальную процедуру – это вариант динамического перехода, аргументом которого является эта формальная метка.

Для случая C, C++ ситуация чуть сложнее. При запуске процедуры из какого-либо объекта приватные данные всех объектов данного типа, которые будет обрабатывать эта процедура, становятся доступными с помощью работы с типом описанной ранее. Но, чтобы это произошло, на спецрегистре процессора должна быть выставлена тегированная величина, содержащая этот тип. По существу – это изменение глобального контекста.

Разумеется, можно было бы всё реализовать по методу формальной процедуры, обсуждённой выше. Но Э3-Е2К используется возможная здесь оптимизация. Дело в том, что тип, выставляемый на регистр, однозначно связан с кодом, на который происходит переход. Поэтому этот тип может выставлять сама запущенная процедура. В результате при запуске контекст переключать нет необходимости, и этот случай сравнивается со случаем запуска глобальной процедуры. Т.е. в момент запуска достаточно иметь только метку перехода.

Почему выставление типа самой процедурой не противоречит защищённости, будет более подробно обсуждено, в разделе *Единица компиляции*.

Для случая указателя на функцию или формальной процедуры предусмотрена тегированная величина (формальная метка), которая в данном случае включает в себя только метку перехода (номер команды).

Как и в общем случае формальной метки, она создаётся в контексте, где доступна эта метка, а затем может свободно передаваться в произвольные контексты, присваиваться в указатели и т.д.

Резюме

Ниже приведены основные операции, реализованные в машинах.

Эльбрус 1, 2

Поддержана статическая вложенность процедур с помощью набора ДИСПЛЕЙ регистров, на которых хранятся указатели на последовательность локальных данных процедур, статически охватывающих работающую.

ОПЕРАЦИЯ ОТКРЫТОГО ВХОДА В ПРОЦЕДУРУ

Для случая, когда новый глобальный контекст – подмножество старого.

Литеральные аргументы – номер команды (информация ему эквивалентная) и номер ДИСПЛЕЙ регистра N из числа активных в месте запуска. Новый глобальный контекст включает ДИСПЛЕИ 0-N.

ОПЕРАЦИЯ ВХОДА В ФОРМАЛЬНУЮ ПРОЦЕДУРУ

Аргумент из регистра – метка формальной процедуры. По контекстной ссылке в метке корректируются, вообще говоря, все ДИСПЛЕЙ регистры нового контекста.

ОПЕРАЦИЯ ФОРМИРОВАНИЯ ФОРМАЛЬНОЙ МЕТКИ

Формируется формальная метка, которая содержит ссылку на контекст, ссылку на код и лексикографический уровень формальной процедуры.

ОПЕРАЦИЯ ВОЗВРАТА

Э3-Е2К

Поддерживается только один уровень общих глобальных данных.
Один глобальный регистр

Поддерживается, также, объектно-ориентированное программирование.
Регистр типа объекта разрешающий доступ к приватным данным всех объектов данного типа.

ОПЕРАЦИЯ ОТКРЫТОГО ВХОДА В ПРОЦЕДУРУ

Все случаи кроме запуска процедуры по указателю и процедур из другой *Единицы компиляции* (см. ниже).

Литеральный аргумент – номер команды.

ОПЕРАЦИЯ ВХОДА В ПРОЦЕДУРУ ПО УКАЗАТЕЛЮ

Аргумент из регистра – тегированный номер команды.

ОПЕРАЦИЯ ФОРМИРОВАНИЯ ТЕГИРОВАННОГО НОМЕРА КОМАНДЫ

Литеральный аргумент – номер команды.
Формируется тегированный номер команды.

ОПЕРАЦИЯ УСТАНОВКИ ТИПА ОБЪЕКТА НА РЕГИСТР

Литеральный аргумент – тип объекта.
Аппаратура проверяет, входит ли данный тип во множество типов данной *Единицы компиляции* (см. ниже).

ОПЕРАЦИЯ ВОЗВРАТА

Все операции переключения контекста, с точки зрения эффективности, имеют аналоги в существующих машинах и допускают эффективную реализацию.

Проблемы стека

При реализации процедурного механизма в Эльбрусе, как и во всех других машинах, для повышения эффективности выделения памяти для локальных данных процедур используется механизм стека.

В Эльбрусе 1, 2 реализован единый стек в памяти, верхушка которого хранится в процессоре на регистрах. Эти регистры адресуются только стековыми операциями.

В Э3-Е2К реализованы окна переменной длины на прямо адресуемых регистрах, работающих как стек и имеющий продолжение в памяти. Кроме того, имеется стек в памяти, как во всех современных машинах.

При выделении памяти в стеке по существу происходит повторное использование памяти, освободившейся от других переменных, что, как уже отмечалось, порождает некоторые проблемы.

Во-первых, в этой памяти может оказаться «мусор», в том числе содержащий ссылки, которые не должны попасть к новому владельцу. Эту память необходимо почистить «пустышками».

Во-вторых, память в стеке, в отличие от других случаев, благодаря стековому принципу пере используется. Поэтому, на эту область могут остаться ссылки у старого владельца и, если не принять меры, то он получит доступ к информации, ему не принадлежащей.

Чистка «мусора» (инициализация выделяемой памяти)

Эльбрус 1, 2

В Эльбрусе 1, 2 аппаратный механизм процедур, выделяя память в стеке для очередной процедуры, прописывал её тегированной величиной «пусто». Массивы обычно генерировались вне стека, и ОС выделяла память под них предварительно почищенную.

Регистровый файл Э3-Е2К

В аппаратуре регистрового файла предусмотрен один бит на каждый регистр. Единица в этом бите означает, что в данном регистре храниться «пусто», не зависимо от реального его содержания. При считывании из него выдаётся «пусто». Любая запись в регистр обнуляет этот бит.

Механизм процедур, выделяя окно в регистровом файле для очередной процедуры, параллельно помечает выделенные регистры этим битом

Таким образом, пока помеченная ячейка на регистрах, бит работает. Затем, либо она прописывается полезной информацией, либо, после откачки/подкачки в память автоматически (при считывании для откачки в память) превращается в настоящую «пустышку».

Стек в памяти Э3-Е2К

Похожее решение реализовано для стека в памяти.

В процессоре существует битовая шкала, когерентно соответствующая верхушке стека в памяти. Как и в регистрах, бит в этой шкале обозначает, что соответствующая ячейка памяти пуста.

Отличие заключается в том, что здесь нет процесса откачки/подкачки, который превращает помеченную ячейку в настоящее «пусто» и обнуляет (освобождает) бит. Поэтому, при приближении шкалы к переполнению, аппаратура сама прописывает наиболее удалённые от вершины ячейки стека «пустышками» и обнуляет (освобождает) биты шкалы.

Указатели на стек

Для решения этой проблемы, в ссылки, смотрящие в стек, вводится поле, обозначающее номер поколения процедур в стеке, на которое указывает данная ссылка.

При записи ссылки, смотрящей в стек, через другую ссылку, смотрящую в тот же стек, аппаратура проверяет, чтобы не образовалась ссылка из более глобального поколения процедуры на данные с меньшим временем жизни.

В результате, в стеке все ссылки смотрят только в тот же стек и только в сторону его дна.

Это значит, что все ссылки будут уничтожены раньше, чем информация, на которую они указывают.

Запрещаются все другие виды ссылок на стек (из любых других областей памяти на данный стек и внутри стека из его более глобальной области на более локальную).

Впервые здесь вводится ограничение, которое будет обсуждено позже.

Единица компиляции

В данном разделе будут обсуждены некоторые важные оптимизации, которые не нарушают описанных принципов, повышают эффективность и полностью сохраняют описанный уровень защищённости.

Описываемая система обеспечивает полную защиту от любых ошибок в программах и/или злонамеренных атак из *других программ пользователей*. Более того, не опасны с точки зрения защищённости и ошибки в значительной части системы. Однако, разумеется, для этого определённая часть системы должна работать правильно. Представляется, что описываемый подход позволяет сократить объём этой критической части до минимума.

В него входят, в основном, компоненты обеспечивающие реализацию всевозможных ссылок и контекстных переключений (процедурный механизм).

Как и в существующих системах, определённый объём программ может быть объединён в единицу компиляции. Единица компиляции – это набор вычислительных модулей

оттранслированный, статически объединённый и готовый к запуску на исполнение. Программы из различных единиц компиляции могут вызывать друг друга, в ходе выполнения единой задачи, работая в единой виртуальной памяти. Такая организация позволяет провести ряд оптимизаций, повышающих эффективность. Здесь будут приведены соображения объясняющие, почему они не нарушают защищённости.

С этой целью будет полезно разобраться во взаимодействии некоторых участников вычислительного процесса.

Вычислительный процесс состоит из модулей или процедур программ пользователей, которые должны быть защищены друг от друга согласно описанным принципам.

Эти модули могут объединяться, быть может, иерархически в более крупные единицы (объекты и т.д.) и, наконец, всё это помещается в единицу компиляции. Существуют программы, которые производят это объединение. Такие, например, как линковщик или компилятор. Будем называть здесь такую программу СБОРЩИКОМ. И затем различные модули из различных единиц компиляции работают и могут взаимодействовать между собой в одной системе. Здесь с ними работают системные программы и аппаратура.

Пользовательские программы объединяются в единицу компиляции СБОРЩИКОМ. В этом факте присутствует акт доверия. У пользовательских программ или их авторов нет никаких возможностей защититься от ошибок или злонамеренных атак СБОРЩИКА, которому эта программа отдана для сборки.

Нет таких возможностей и у системы. Бессмысленно, например, предусматривать защиту с помощью аппаратуры или системных программ от ошибок или злонамеренных атак СБОРЩИКА приводящих к нарушению защищённости между модулями, данными ему для сборки. Это сделать просто невозможно.

Поэтому, для пользователя единственной возможностью здесь остаётся правильно выбирать СБОРЩИКа и затем считать, что СБОРЩИК не содержит ошибок, приводящих к нарушению защищённости между пользовательскими программами, переданными ему для сборки. Это вполне естественно, так как разработчики отдельных модулей добровольно передали свои программы СБОРЩИКУ для сборки. Это и есть акт доверия.

Можно было бы предположить, что любой СБОРЩИК не нарушает защищённости и между различными единицами компиляции. Но тогда его следовало бы включить в число системных программ. Это не соответствует принятой практике и совсем не является необходимым.

Любой СБОРЩИК это обычная пользовательская программа. В ней можно предпринимать все возможные меры для оптимизации и повышения эффективности результирующего кода. Пользователи сами ответственны за выбор качественного СБОРЩИКА.

Для разработчиков отдельных модулей, СБОРЩИК - это система, а для системных программ, СБОРЩИК - это рядовая пользовательская программа.

Задача обеспечения защищённости между различными единицами компиляции ложится на плечи системы и аппаратуры.

Этой цели служат описанные здесь аппаратные функции. В качестве примера системной программы, поддерживающей защищённость **между** единицами компиляции можно привести верификатор для литеральных констант операции CAST, описанный выше.

Эту работу нельзя поручить СБОРЩИКУ, так как его ошибка (неправильная литеральная константа CAST) может позволить программе пользователя выйти за границы объекта и нарушить работу программы из другой единицы компиляции.

В разделе *Переключение контекста* его реализация описывалась в том виде, как она выполнена в Э3-Е2К, уже с учётом оптимизаций, внутри единицы компиляции, с обещанием обсудить их законность в данном разделе. Это обсуждение сводится к следующему.

Введение единицы компиляции позволяет использовать четыре оптимизации.

1. Литеральные переходы между всеми процедурами внутри данной единицы компиляции. Разумеется, если эти запуски не являются динамическими по существу (запуск процедуры по указателю, виртуальные функции и т.д.)

2. Установка типа объекта внутри самой процедуры объекта.

3. Размещение статической части типа в общих глобальных данных.

4. Реализация процедур in line.

Рассмотрим, сначала, как с точки зрения защищённости должна была бы работать система без всех этих оптимизаций. Будем, в основном обсуждать случай Э3-Е2К, в наибольшей степени использующий концепцию единицы компиляции.

Предположим, что оттранслированные программы отдельно для каждого типа (класса в С++) (отдельная трансляция каждого типа несколько противоречит наличию общих глобальных данных в языке С++, например, но здесь на время игнорируется этот факт) поступают в системную программу загрузки типа (class loader), которая предполагается отлаженной, так как она работает со ссылками и распределением памяти. Можно рассматривать переведение её значительной части в статус пользовательской, но этот шаг значительно и без разумных оснований усложнил бы аппаратуру.

Эта системная функция загружает все программы в виртуальную память и формирует все тегированные величины (как правило, они содержат виртуальные адреса). Она же формирует статическую область данного типа.

В статическую область помещаются:

- тегированные метки всех процедур (методов) данного типа. Каждая метка состоит из номера команды данной процедуры и виртуального адреса данной статической области, которая будет служить глобальным контекстом для каждой из процедур (Реально, как будет описано в разделе *Общие глобальные данные*, здесь используется упрощающая оптимизация).

- тегированный код данного типа, который каждая процедура должна будет перегрузить в спецрегистр, открывающий доступ данным процедурам к приватным данным любого объекта этого типа.

- тегированные константы для операций преобразования типа данного объекта к его базовым типам (CAST).

- тегированная «болванка» ОД данного типа, содержащая все поля ОД данного типа, кроме виртуального адреса объекта, который добавляется в ОД в процессе генерации объекта.

- во все оставшиеся ячейки помещается либо начальные значения (инициализация), предусмотренные программой, либо тегированная переменная «пусто», вызывающая диагностическое прерывание, при попытке использовать её в операции.

Так как предполагается, что каждый тип оттранслирован отдельно, то следует так же предположить, что статическая область содержит ссылку на общие глобальные данные. Глобальные данные, кроме глобальных переменных, предусмотренных языком, содержит все глобально доступные по синтаксису языка метки процедур. Эта ссылка должна помещаться на регистр глобальных данных механизмом процедур при входе в любую процедуру этого типа. Предполагается, так же, что метки процедур будут выдаваться «наружу» на уровне системы файлов в соответствии с некоторой процедурой «знакомства».

Нетрудно убедиться, что такая схема полностью соответствует требованиям защищённости, представленным выше.

Нетрудно, так же, видеть, что она заметно менее эффективна, чем схемы, используемые в существующих машинах. В основном понижается эффективность вызова процедуры.

Теперь нетрудно оценить возможности оптимизации, в связи с введением единицы компиляции.

При использовании единицы компиляции, загрузчик работает почти так же, с той лишь разницей, что все статические области всех типов, объявленных в данной единице компиляции вместе с глобальными данными помещаются в единую область глобальных данных единицы компиляции.

Литеральные переходы

При правильной работе СБОРЩИКА достаточно очевидно, что при замене вызова процедуры через метку на литеральный вызов, работа программы не измениться.

Если СБОРЩИК это компилятор, то для него внутренний вызов ни чем не будет отличаться от простого перехода, который всегда транслируется литерально.

Даже если некоторые процедуры написаны на ассемблере Э3-Е2К (защищённого режима) это не меняет ситуации благодаря специальной организации системы команд Э3-Е2К, которая позволяет СБОРЩИКУ полностью проконтролировать ассемблерный текст.

В системе команд Э3-Е2К нет динамических переходов по произвольному целому, которые помешали бы провести анализ кода. Это сделано не в угоду защищённости, а просто потому, что они вовсе не нужны для трансляции любых конструкций любых языков.

Есть команда SWITCH с контролем границ. Есть команда динамического запуска процедуры, которая работает по тегированной величине. Формируется эта величина командой из литерала, так что его значение может быть проконтролировано статически, при этом аппаратура следит, чтобы значение не выходило за границы единицы компиляции и т.д.

Таким образом, любой ассемблерный текст может быть полностью статически разобран.

Благодаря этому, любая ассемблерная программа может быть проверена СБОРЩИКОМ на правильность литеральных переходов и на правильность переходов внутрь этой программы.

Все внешние вызовы будут происходить через тегированные метки, которые так же находятся под контролем СБОРЩИКА.

Установка типа внутри процедуры

Речь идёт о том, что тип объекта, к которому принадлежит запускаемая процедура, выставляет на спецрегистр сама процедура литерально из кода в первой команде процедуры.

Здесь так же достаточно очевидно, что, при правильной работе СБОРЩИКА замена установки типа из формальной метки на литеральную установку не изменит работы программы.

Ассемблерная программа здесь так же допустима, так как благодаря возможности статически разобрать код СБОРЩИК сможет её проконтролировать на правильность выполнения этого требования.

При работе операции установки регистра типа из литерала, аппаратура контролирует, чтобы выставляемый тип был бы из диапазона типов, принадлежащих данной единице компиляции.

В данном случае доверие правильной работе СБОРЩИКА не достаточно, так как установка типа, принадлежащего другой единице компиляции, может нарушить правильную работу **других** программ.

Общие глобальные данные

Объединение всех статических переменных вместе с глобальными данными в один пул в принципе рождает проблему, так как из любой процедуры любого типа доступны статические переменные всех типов.

И здесь, как и выше, при отлаженном СБОРЩИКЕ и коде, допускающем статический разбор, проблема решается. Можно надёжно проверить даже для ассемблерного кода, что код каждой процедуры обращается только к дозволенным ему данным.

Не исключается возможность иметь в глобальной области массивы. При формировании указателя на этот массив его границы должны быть указаны литерально, что позволяет вести контроль. Аналогично можно работать через указатель и со скалярными переменными.

Ассемблер, как и в предыдущих случаях не составляет исключения.

Процедуры in line

При реализации допускается подставлять код процедуры AA одного типа A in line в код процедуры BB другого типа B, вместо вызова AA из BB, если оба типа принадлежат одной и той же единице компиляции.

Как и для обычной процедуры, в подставленной процедуре AA до первого обращения к приватным данным её объекта типа A должна быть выполнена команда загрузки спецрегистра процессора кодом типа A литерально из команды. В конце подставленной процедуры AA после всех обращений к объекту типа A значение спецрегистра типа должно быть восстановлено и в него должен быть возвращён код типа B. При этом допускаются обычные оптимизации.

Следует заметить, что в архитектуре широкого командного слова Э3-Е2К, эти команды перезагрузки практически никогда не будут вызывать затраты лишних тактов машины, так как в местах вызовов процедур велика вероятность найти свободный слог в команде (об архитектуре Э3-Е2К см. [15]).

Правомерность этого при тех же предположениях (СБОРЩИК отлажен, код можно разобрать статически) достаточно очевидна, в том числе с учётом ассемблера.

Аппаратная поддержка единицы компиляции

В отличие от существующих систем, в Э3-Е2К в одном виртуальном пространстве могут работать несколько единиц компиляции взаимодействующие между собой.

Система поддерживает таблицу единиц компиляции, которые входят в определённое виртуальное пространство.

Для каждой единицы компиляции в этой таблице хранится:

- диапазон виртуальных адресов, в котором находятся все её программы,
- диапазон кодов типов объектов, определённых в данной единице компиляции,

- ссылка на её область глобальных данных.

В процессоре существуют соответствующие спецрегистры, которые хранят эти три величины для работающей единицы компиляции.

Кроме того, как уже упоминалось, существует спецрегистр, хранящий код типа объекта, процедура которого выполняется в данный момент.

При всех литеральных переходах и вызовах процедур аппаратура проверяет, что этот переход не выходит за границы единицы компиляции.

При всех литеральных загрузках кода типа работающего объекта, с помощью регистра диапазона типов, аппаратура проверяет, не выходит ли он за диапазон типов, объявленных в данной единице компиляции.

В таблице страниц виртуального пространства для каждой страницы, содержащей программы, хранится номер единицы компиляции, к которой эта страница относится.

При динамическом переходе по тегированному номеру команды (т.е. в случае, когда переход пересекает границу единицы компиляции), аппаратура автоматически загружает указанные выше спецрегистры и обнуляет регистр текущего типа.

Эффективность

В представленном выше материале достаточно подробно описаны основной подход и реализация технологии защищённости Эльбрус, реализованной в трёх поколениях машин.

Но, разумеется, хотя для обеспечения выполнения поставленных задач и реализации существующих языков в Эльбрусе сделано всё, здесь описано не всё.

Нет описания таких, достаточно важных для некоторых языков черт, как SET_JUMP/LONG_JUMP, MEMBER POINTER, FRIENDS, виртуальные методы и т.д.

Однако думается, что приведённого здесь вполне достаточно для ясного представления того, что всё это реализуемо. Приведённого достаточно и для того, чтобы показать характер реализации. Кроме того, описаны все решения, затронувшие аппаратуру.

Несколько слов об эффективности.

Введённые изменения касаются в основном операций обращения в память и вызова процедур. Остальные операция почти не испытали влияния введения защищённости.

Поэтому практически нет никаких ограничений на использование оптимизаций, применяемых в подобных случаях в связи с защищённостью.

Тем не менее, за новое свойство всё-таки приходится платить.

Наиболее серьёзное влияние на эффективность оказывает введение 128 разрядных указателей. Этот факт увеличивает объём требуемой памяти и требует дополнительных ресурсов исполнительных устройств при выполнении операций.

Ситуация несколько облегчается в связи с тем, что в современных машинах увеличивается параллельность обработки.

К слову сказать, все мы являемся свидетелями перехода в последнее время на существующих машинах с указателей формата 32 бита на 64 бита. Этот переход не вызвал сколько ни будь серьёзных дискуссий в связи с его влиянием на эффективность. Более того, достаточно распространённая в настоящее время система IBM AS/400 уже много лет использует указатель формата 128 бит.

Другое снижение эффективности связано с введением новых команд. Это, прежде всего адресные преобразования (CAST, например) и установка типа объекта.

Некоторые операции потребовали большего времени для своего исполнения.

На операции обращения в память (доступ в память), несмотря на их усложнение, это не сказалось или почти не сказалось. Но команда запуска процедуры потребовала несколько большего времени, чем простой переход.

Сложность оборудования, безусловно, возросла. Влияние на его объём можно грубо оценить как, в основном, добавление тегов во все регистровые памяти кристалла (2 бита на каждые 32), приблизительно 6-7% от объёма регистровых памятей кристалла.

Следует отметить, что хорошая реализация защищённости, позволяет в определённой части системы повысить эффективность. Для объяснения этого положения обсудим прежде некоторые аспекты существующих систем.

Ввиду практически отсутствия защиты внутри виртуальной памяти, в существующих системах каждая независимо оттранслированная программа исполняется в отдельном виртуальном пространстве. Это обеспечивает защиту между этими программами.

Однако такое решение не лишено недостатков, приводящих к неудобству в программировании и потере эффективности.

Во-первых, независимо откомпилированные программы, работающие в различных виртуальных пространствах, не могут эффективно и удобно работать над общими данными. Нельзя, например, передавать из одной программы в другую параметры по ссылке и т.д. Возможность использовать общие страницы не решает этой проблемы или решает её не достаточно удобно и эффективно.

Во-вторых, нельзя полностью избавиться от необходимости работать в одной виртуальной памяти независимо откомпилированным программам.

Вместе с программой пользователя в той же виртуальной памяти должна работать ОС и библиотеки. При этом процедуры ОС используют отдельный стек, а библиотеки и

пользовательская программа, написанные разными авторами, плохо или совсем не защищены друг от друга.

Всё это усложняет систему, усложняет её использование, совсем не улучшает надёжность и плохо справляется с проблемой защищённости.

В Эльбрусе, благодаря решению проблем защищённости, нет ограничения на работу различных независимо откомпилированных программ в единой виртуальной памяти. Любая процедура может вызвать любую другую в той же виртуальной памяти, на том же стеке, вне зависимости от того, в какой независимо откомпилированной программе она находится.

Другой стек рождается только тогда, когда необходима действительно параллельная деятельность. Другое виртуальное пространство рождается только при запуске полностью независимой работы.

Поэтому с точки зрения методов запуска и работы программ нет различий между программами пользователя, библиотеками и ОС.

Всё это, безусловно, положительно сказывается на эффективности и простоте пользовательских и системных программ.

ОГРАНИЧЕНИЯ

Как уже отмечалось, на Эльбрусе 1, 2 были реализованы большинство распространённых в то время языков. Языки были реализованы в достаточно полном объёме, и не потребовалось вводить каких бы то ни было ограничений.

В то время объектно-ориентированная технология только начинала использоваться, не было C++ в широком использовании. Но в рамках базового языка высокого уровня Эльбрус Эль-76 была реализована поддержка абстрактных типов данных.

На Э3-Е2К реализованы C, C++, ФОРТРАН, просмотрена реализация JAVA.

Пришлось ввести три ограничения для языка C++, два из которых относятся и к C. Эти ограничения, связаны с тем, что ограничиваемые черты прямо противоречат защищённости.

Более того, выполнение всех этих ограничений улучшают стиль программирования, повышают эксплуатационные характеристики программ, могут рассматриваться скорее как исправление ошибок в определении языков, и вряд ли вызовут критику у большинства программистов.

Присвоение целого в указатель

Это свойство совсем не трудно реализовать совместимым образом, но его противоречие защищённости достаточно ясно. Чтобы быть совместимым с этим свойством нужно уметь исполнять программы, которые прямо нарушают защиту памяти.

Можно было бы разрабатывать какие-либо другие трактовки этого свойства. Но их реализация всё равно требовала бы введения некоторого ограничения и не привела бы к сохранению совместимости.

Для оценки ущерба от введения этого запрета, были изучены практические случаи его применения, и стало достаточно очевидно, что, во-первых, они весьма малочисленны и, во-вторых, всегда можно найти альтернативный способ программирования, используя только защищённые черты языка и без заметной потери эффективности.

Как уже отмечалось, современные компиляторы выдают предупреждение в случае использования этого свойства, что явно свидетельствует о его постепенном исключении из практики программирования.

Переопределение оператора “new”

Это свойство так же не представляет труда реализовать точно, но оно прямо приводит к нарушению модульности, секретности, защищённости и т.д.

Представьте, что с помощью этого свойства, программист А создаёт объект, тип которого разработан программистом В и при этом переопределяет оператор “new” таким образом, что этот объект создаётся на массиве программиста А. В результате А будет иметь неограниченный доступ ко всем приватным данным объекта, хотя и созданного по его заказу для решения его задачи, но разработанного программистом В.

Это может привести к неправильной работе программ В, доступу к некоей секретной информации, которую В не собирался разглашать и нарушению защиты памяти.

Ссылки, смотрящие в стек

Как уже обсуждалось выше, проблема возникает, если в результате выполнения программы возникает ссылка в стек из области, более глобальной, т.е. имеющей большее время жизни.

Как, например, ссылка на стек из глобальной области или из другого стека, или ссылка из области, расположенной ближе к дну стека, на область, расположенную ближе к его вершине.

В этих случаях, из-за переиспользования виртуальной памяти в стеке, программа через зависшие ссылки может получить доступ к недозволенным данным.

Если ссылка в стек направлена от более локальной области в более глобальную, проблем не возникает, так как по мере возврата из процедур ссылка исчезнет раньше, чем будет переиспользована область стека, на которую она указывает. Однако в других случаях, в том числе, когда ссылка адресует стек снаружи, проблема остаётся.

Этой проблемы нет практически во всех языках, но она существует в C и C++, т.е. в случае, наиболее важном, в практическом смысле.

В этих языках такое программирование допустимо и существующие реализации языка не контролируют возможное нарушение защиты в результате этого.

Другими словами, так же, приблизительно, как и с записью целого в указатель, программа нарушающая защиту, должна считаться правильной C или C++ программой.

Здесь, так же, необходимы ограничения.

В принципе, существуют две возможности: либо разрешать любые варианты ссылок, но контролировать появление зависших указателей, либо разрешить ссылки только в безопасном направлении.

Первый вариант труден и несколько неэффективен в реализации, более того он формально не обеспечивает совместимости. Всё равно необходимо вводить ограничение, правда, более очевидное. Однако, к сожалению, довольно много программ используют такие «неудобные» ссылки.

Поэтому в Эльбрусе Э3М-Е2К реализованы оба варианта.

Кроме того, следует отметить, что введение этого ограничения в значительно большей степени соответствовало бы современному пониманию хорошего и правильного стиля программирования.

Проблема совместимости

При разработке защищённой системы перед разработчиками возникает следующая не простая проблема.

Дело в том, что достижение защищённости находится в очевидном противоречии с задачей обеспечения эффективной совместимости.

Это происходит, по крайней мере, по двум причинам.

Во-первых, само определение наиболее распространённых языков C и C++ не полностью соответствует требованиям защищённости в той части, которая была обсуждена выше.

Во-вторых, выполнение не всех требований языка контролируется реализацией (например, выход за границы массивов или использование неинициализированных данных), в результате, реальные программы, обычно, содержат ошибки, в том числе, приводящие к нарушению защищённости, которые обнаруживаются системой Эльбрус.

Практика переноса программ на Эльбрус показывает, что практически во всех сколько-нибудь солидных программах, при их исполнении на Эльбрусе обнаруживаются ошибки.

Проблема, следовательно, заключается в том, что защищённая система не может быть совместима с существующими реализациями с точностью до нарушений защищённости.

В связи с этим, в Эльбрусе реализованы две или даже три возможности.

Реализована технология полной двоичной совместимости с архитектурой Интел x86.

Исполнение программ в этом режиме в точности соответствует семантике двоичного кода, с точностью до незащищённых черт внутри виртуального пространства, в котором решается задача.

Существует защищённый режим, в котором исполняются программы, оттранслированные с языков, с учётом обозначенных ограничений. В этом режиме выполняются все требования защищённости.

Наконец, существует смешанный вариант, когда защищённая программа может, например, вызывать незащищённые библиотеки.

При этом задача будет работать в двух пространствах: защищённом и незащищённом.

В незащищённом пространстве не будет гарантирована защищённость между незащищёнными компонентами задачи. Все же защищённые компоненты будут защищены, как друг от друга, так и от незащищённых частей.

Эльбрус обеспечивает переход к защищённости, но это *переход*. Всякий, кто собирается решать задачи на Эльбрусе в защищённом варианте, должен быть готовым к этому переходу, т.е. к исправлению ошибок в программе, противоречащих защищённости.

Важной целью, с точки зрения защищённости и совместимости достигнутой в Эльбрусе, представляется обеспечение полной обратной совместимости.

Программы, отлаженные на Эльбрусе, смогут, безусловно, выполняться на других системах (разумеется, после перетрансляции). Но они уже будут содержать значительно меньшее число ошибок, в том числе, будут исключено значительное число ошибок нарушающих защищённость, в особенности, если эти программы достаточно долго проработали на Эльбрусе.

ФАЙЛЫ

Файлы являются другой частью системы, кроме оперативной памяти, в которой располагается информационное пространство решаемой задачи. Поэтому все проблемы, возникшие и решённые для оперативной памяти, должны быть обсуждены и в контексте системы файлов.

Казалось бы, одним из возможных решений могло бы стать слияние системы файлов с системой оперативной памяти. В прошлом были попытки осуществить такой шаг. Наиболее известные из них – это система Малтикс и IBM S/38 – AS 400.

Стимул для этих попыток достаточно ясен. Единое информационное пространство – это, концептуально, более простая система, чем система с двумя пространствами: оперативной памятью и файлами. Рост объёма виртуальной памяти так же способствовал росту интереса к такому слиянию. Казалось бы, объём виртуальной памяти может вместить и файлы.

Загрузка файлов в виртуальную память – это несколько другой, более технический вопрос и его можно решать независимо от более концептуального вопроса: «можно ли и нужно ли объединять файлы и память, как концепции, в единую?».

До сих пор это не удавалось выполнить практически. Однако это нельзя считать доказательством бесперспективности этого направления.

Достаточно вспомнить историю внедрение страничной виртуальной памяти. Первые университетские работы появились в конце 50х, и до введения системы IBM 370 в середине 70х, страничную виртуальную память можно было бы считать бесперспективной.

Однако представляется, что с файлами ситуация складывается другая. По всей видимости, существуют объективные причины, которые вызывают необходимость поддерживать два разных информационных пространства - файлы и виртуальную память.

Попытаемся разобраться в этих причинах, с риском высказать несколько спорные соображения.

Совсем кратко разница между объектом в виртуальной памяти и файлом заключается во времени жизни. Время жизни объекта в виртуальной памяти, как правило, конечно, и ограничено временем работы какой-либо откомпилированной программной единицы. Файл имеет неопределённое время жизни, выходящее за границы работы любой программной единицы.

Несколько более детально.

Для объектов памяти события происходят в следующей последовательности.

Вначале полностью составляется программа, в которой объявляется, и по заказу которой создаётся объект в памяти. Ещё до начала её исполнения в программе определены все обращения к этому объекту и известно, что объект прекратит своё существование вместе или до конца работы программы.

Такая ситуация позволяет провести трансляцию программы с оптимизацией. Она иногда влияет и на написание программы программистом. Лишь после полного завершения написания и трансляции оптимизированная программа начинает работать, создаётся объект, с ним ведётся работа и он прекращает своё существование.

В случае с файлами, файл, вообще говоря, создаёт программист. В момент его создания совсем не известно, какие программы с ним будут работать и когда он перестанет существовать.

Одним из примеров влияния такого рода глобализма файлов является различие в методах синхронизации доступа к данным в памяти и в файлах. В случае с памятью, программист видит всю программу до начала её исполнения, в том числе видит все критические секции по работе с общими данными при многопроцессорной обработке. Это позволяет программисту правильно выбрать множество семафоров.

В случае файлов, в любой момент времени, полностью отсутствует информация о том, какие программы будут работать с этим файлом в будущем. Именно по этому в системе файлов для синхронизации используют не семафоры, а технику сеансов. В памяти, сеансы использовать нерационально, более эффективно использовать семафоры, которые, по существу, являются оптимизацией техники сеансов.

Другим примером может служить способ адресации. В памяти используется «оптимизированная цифра» - виртуальный адрес. В системе файлов – символьные имена, так как одна из «программ», работающая с файлами – это программист за дисплеем, которому лучше понятны символьные имена в виде строк.

Приведённые выше рассуждения наводят на мысль, что, по крайней мере, в настоящее время, не целесообразно надеяться на успешное исключение понятия файлов из практики программирования. В ОС Эльбрус 1, 2 таких попыток не делалось.

Это, разумеется, не касается вопроса использования, доступного в настоящее время, огромного виртуального пространства для повышения эффективности работы системы файлов.

Традиционные системы файлов

Постараемся теперь обсудить, как обстоит дело с защищённостью в системах файлов существующих ОС.

Для этого, прежде всего, кратко опишем основы организации систем файлов существующих ОС. Эти положения всем хорошо известны, но здесь полезно их привести, чтобы выделить области наиболее важные для обсуждаемого вопроса.

Рассматриваемые ниже вопросы имеют очень близкие по своему характеру решения практически во всех распространённых в настоящее время ОС, таких как Windows, Linux, Solaris, HP-UX, ОС AS/400 и т.д.

Именование файлов

Именование файлов в существующих системах достаточно просто. Каждый файл имеет символьное имя в виде строки. Имя может быть многослоговое, со слогами, разделёнными слешами. Общее пространство файлов – это дерево с единым для данной машины корнем.

Каждому слогу соответствует справочник. Последний справочник указывает на файл.

Все пользователи, работающие на машине, имеют доступ к общему корню.

Имя, начинающееся со слеша – это полное имя от корня. Многослоговое имя, начинающееся со слога – это относительное имя, от текущего справочника. Специальный слог в относительном имени позволяет подняться вверх по дереву именования.

Для имени, состоящего из одного слога, каждый пользователь может снабдить систему списком имён справочников, в которых следует искать этот файл.

Так как для представления имени файла в языках программирования используется строка символов, никаких специальных типов данных для этой цели не вводится. Любая программа в любое время может попытаться обратиться по имени к любому файлу на данной машине.

Безусловно, положительной чертой этой системы является простота её организации.

Кажется, она не слишком усложнилась по сравнению с первыми пакетными системами, где файлы искались по символьному имени в колоде перфокарт или на магнитных лентах.

Защита файлов

В существующих системах файлов, для обеспечения защиты используется классическая ключевая система.

ОС для каждого файла хранит имя пользователя или группы пользователей, которым разрешён доступ к этому файлу, вместе с правами доступа (читать, модифицировать и т.д.).

Такая система защиты, так же отличается простотой организации.

Анализ

Ещё раз полезно отметить основные требования к качественной системе защиты.

Это, прежде всего, надёжность. Система защиты должна разрешать доступ данной программе к минимально возможному объёму информации, который, в то же время, должен включать всё необходимое.

Кроме того, и это очень важно, система защиты, при всей жёсткости, должна быть простой, не только и не столько в организации, но, прежде всего, в использовании.

Подавляющая масса программистов, не выражает какой бы то ни было конкретной критики к существующей системе, так как при конкретном программировании они озабочены другими проблемами, отличными от проблем защищённости.

Если требования защиты создают какие-нибудь неудобства, то их чаще всего можно устранить, расширив область разрешённого доступа. Иной раз, даже при существующей системе можно улучшить качество защиты, но при этом необходимо, что-то сделать. Часто это не делается.

Причина вполне объяснима. Вероятность того, что именно это действие отвратит беду, исчезающе мала, неудобства же и возможная потеря эффективности очевидны.

Качественная система защиты должна обеспечивать надёжный доступ к **минимально** необходимому объёму и не требовать от программиста **ни каких** дополнительных усилий. Программист в первую очередь заботится о том, чтобы правильно решалась задача. О защите вспоминают, когда случается беда.

Поэтому следует проанализировать, в том числе и в большей степени именно сложившиеся способы использования рассматриваемой системы, а не теоретически возможные, но не применяющиеся на практике, ввиду их неудобства и неэффективности.

Именно с такой точки зрения постараемся провести анализ.

При запуске задачи на счёт в своём задании программист указывает имена файлов. Первый шаг, запуск своей программы на счёт не нуждается в комментариях, пользователь ограничен в этом набором своих собственных файлов. Но даже в этом случае работа самой запущенной программы требует комментариев.

Запущенная программа имеет доступ ко всем файлам её владельца, хотя для своей работы обычно ей необходим доступ до значительно меньшего их числа или не нужен доступ до файлов вообще. При ошибке в программе или, если программа инфицирована какой-либо злонамеренной программой, это может привести к проблемам.

Рассмотрим теперь несколько более сложные случаи.

Первый пример

Программист ПА запускает программу ФВ пользователя ПВ, к которой ему разрешён доступ и передаёт ей в качестве параметра свой файл ФА. Эта программа ФВ в процессе своей работы запускает другую программу ФВ1 пользователя ПВ и передаёт ей в качестве параметра файл ФА, переданный ей как параметр.

Программа ФВ «статически», т.е. по тексту самой программы, а не через параметры, связана с программой ФВ1. Можно сказать, что программа ФВ имеет в своём статическом контексте программу ФВ1.

Схема разрешения доступа:

ПА - > ФА, ФВ
ПВ - > ФВ, ФВ1

Пользователь ПА пускает программу:

ФВ(ФА).

Программа ФВ:

……..CALL ФВ1(<ФА>)…….

Программа ФВ1

… … … READ (<ФА>)… ….

В таком варианте задача будет считаться под именем пользователя ПА и возникает проблема доступа к файлу ФВ1.

Простейшим решением будет разрешить этот доступ пользователю ПА. Но это будет чрезмерное расширение его прав. Не исключено, что пользователь ПВ не желает, чтобы ПА имел информацию об его содержании, а тем более ПА не должен иметь возможность модифицировать ФВ1, что может исказить работу программы ФВ.

Более того, если программа ФВ достаточно популярна и ею позволено пользоваться большому число пользователей, доступ к ФВ1 станет ещё более серьёзной проблемой.

Другой возможностью может быть использование функции SETUID. Если при запуске программы ФВ использована эта функция, то программа ФВ будет работать под именем её владельца ПВ и доступ к файлу ФВ1 не представит проблемы.

Однако в этом случае проблему представит обращение программы ФВ к переданному ей параметру ФА, так как в исходной схеме ПВ не имеет к нему доступа.

Простейшим решением будет создание новой группы ПА+ПВ с разрешением её членам доступа к ФА. Однако если ПА запускает ФВ с различными своими файлами в качестве параметра и, если многие пользователи пожелают работать с ФВ, так же со многими своими файлами, то здесь возникает несколько замечаний.

Во-первых, неприятно, что кроме передачи самого параметра, пользователю приходится создавать группу, давать разрешение, убирать всё это в конце работы и т.д.

Во-вторых, вместе с возможностью работать программе ФВ, пользователь ПВ получает независимый доступ к ФА, что совсем не является необходимым и ослабляет защищённость.

В-третьих, если пользователь ПА по ошибке или по злому умыслу передаст в качестве параметра вместо имени своего файла, строку представляющую имя файла пользователя ПВ,

и если этот файл параметр передан с правом модификации, то безусловные неприятности возникнут для пользователя ПВ.

Второй пример

Другой пример связан с попыткой пользователя П включить в число своих файлов некоторый новый чужой не проверенный файл, который может содержать ошибки или даже злонамеренные атаки. Например, такой файл может быть считан из сети. Пользователь П может даже не включать этот файл в число своих файлов, а просто однократно его исполнить. При этом новому файлу (программе) необходимо передать один файл параметр.

Разумное поведение в этом случае заключается в следующем.

Необходимо создать нового пользователя «ГОСТЬ» - Г и разрешить ему доступ к единственному файлу, переданному ему в качестве параметра. В конце работы необходимо удалить этого пользователя и выданное разрешение. Это не только неудобно, но и неэффективно.

В виду сложности процедуры, типичное поведение здесь состоит в следующем.

Пользователя Г не заводят, а задачу пускают от имени пользователя П, разрешая ему, тем самым доступ к файлу параметру, ну за одно и ко всем файлам доступным П. Действия наименее обременительные с точки зрения пользователя, но крайне рискованные.

Таким образом, если принимать во внимание не теоретические возможности системы, а сложившуюся практику её использования, с учётом удобства и эффективности, то следует заключить, что не решены, по крайней мере, две проблемы: проблема передачи параметра – файла и проблема запуска «ненадёжной» программы. Вторая проблема – это основная проблема любой системы защиты.

Обе эти проблемы, отчасти, связанны между собой. Так как в рассматриваемых системах имя файла – это обыкновенная строка, передача этого имени в качестве параметра «не видна» системе. Более того, часто передаётся не всё имя, а его некоторая часть. Используя эту часть, программа, получившая её в качестве параметра, преобразует её в настоящее имя, в соответствии с соглашениями, известными только запускаемой и запускающей программам.

Система не имеет ни какой информации обо всём этом. Поэтому она не может без помощи программистов разрешить доступ к файлу, переданному в качестве параметра. Для того, чтобы это оказалось возможным, передаваемый параметр – файл должен быть явно обозначен как специальный тип данных. Тогда система автоматически обеспечила бы доступ к нему (и быть может только к нему) запущенной программе.

В рассматриваемых системах этого нет. Поэтому для доступа к файлу – параметру запущенная программа получает доступ к корню дерева имён и право работать от имени пользователя передавшего параметр, т.е. запустившего эту программу.

В результате запущенная программа получает доступ ко всем файлам пользователя. И это происходит, даже если программа считана из сети и к ней нет ни какого доверия.

В существующей системе, вообще говоря, можно обеспечить требуемый уровень защищённости, но часто путём невероятных не оправданных с точки зрения пользователя усилий и заметной потерей времени и эффективности. Поэтому реальная практика это отвергает и использует простые приёмы, не выдерживающие критики с точки зрения защищённости.

По существу у пользователя почти всегда существует дилемма выбрать надёжный или более удобный и эффективный вариант. Но защищённые варианты часто столь неудобны, что они не воспринимаются пользователем как возможные варианты.

Выбор всегда очевиден в пользу удобства. Пользователь интуитивно (и вполне справедливо) уверен, что усилия, потраченные на защищённость в данном конкретном месте, с нулевой вероятностью предотвратят реально возникшую проблему именно здесь. Проблема возникнет где-то в другом месте. А раз так, то зачем тратиться. И защищённость всегда страдает.

В связи с рассматриваемой системой следует отметить ещё одно обстоятельство.

При поиске файла по слоговому имени система использует справочники, управляемые пользователем. Пользователю дана возможность изменить структуру системы справочников, в результате чего имена файлов изменят свой смысл.

Это может создать неразбериху, если, например, слоговые имена включены в тексты программ.

Попытка перенести, например, программный комплекс, состоящий из нескольких ссылающихся друг на друга файлов с «зашитыми» в программу именами, может сопровождаться заметными неудобствами.

Вирусы в существующих системах

Однако наиболее серьёзным вопросом в связи с организацией системы файлов на современных машинах, является проблема борьбы с опасностью вирусов и других злонамеренных атак. Будем называть такие программы агрессивными (АП).

Много полезной информации по этому вопросу можно найти на сайте Касперского [16].

Полезно привести здесь некоторые типичные приёмы АП, не претендуя на какую бы то ни было полноту изложения. Да в этом и нет необходимости, так как, как будет видно из дальнейших разделов, описывающих подход Эльбруса, защита от АП в Эльбрусе – это не борьба с конкретными приёмами авторов АП, но скорее создание системы, обеспечивающей защиту от любых АП.

Типичная последовательность действий АП можно представить следующими шагами.

1. Проникновение в машину пользователя.
2. Достижение цели – быть запущенной.
3. Агрессивные действия.

Проникновение в машину пользователя

Существуют, по крайней мере, три возможности проникновения, часто используемые авторами АП: через перенос программ любыми способами самим пользователем (съёмные носители, сеть), через почтовую систему сети и через служебные программы сети.

В самом этом шаге пока нет ни какой «криминальной» компоненты, ходя, конечно, введённые программы уже содержат проблемы.

Запуск агрессивной программы

Этот шаг, по всей видимости, наиболее трудный для авторов АП и требует большого «творчества».

Если программа была перенесена самим пользователем, то она рано или поздно будет запущена им самим.

Более трудно добиться запуска программы переданной по почте. Однако этот способ более «эффективен», по всей видимости, для авторов АП, так как значительно проще обеспечить массовость.

Существует наивный способ добиться запуска АП – попросить самого пользователя запустить исполняемую программу, приложенную к письму.

Другие способы добиться запуска, используют ошибки в системных программах или в самом дизайне этих программ.

Типично в письме просят посмотреть картинку, приложенную к нему и т.д. При просмотре картинки, благодаря использованию ошибок или по логике программы просмотра картинки, системные программы передают управление на программу, введённую вместе с картинкой.

Известны, например вирусы, использовавшие макро в системе Word, поставляемые автором АП.

Другой пример - использование обменов данных в стек, которые забивают возврат из системной процедуры адресом программы, присланной автором АП. Чуть-чуть более детально этот приём сводится к следующему. Автор АП знает, что система для эффективности считывает некоторую информацию (например, почтовый адрес) прямо в стек. Используя этот факт, автор АП предъявляет фиктивный, специально составленный «адрес» весьма большого размера. Эта считанная информация «забивает» адрес возврата из системной процедуры, который расположен в стеке смежно с областью ввода, адресом агрессивной программы. Эта АП программа находится в составе той же введённой информации. В результате системная процедура «возвращается» в АП программу.

Представляется, что, несмотря на использование ошибок в системных программах, сам по себе запуск любой «чужой» программы будь это даже АП, не должен вызывать аварии в системе, если она хорошо спроектирована с точки зрения защиты.

Глобальный контекст этой программы, если он существовал, остался на машине её автора, и нет никакого логического смысла или разумной пользы предоставлять считанной программе в качестве глобального контекста что-либо из файлов машины, где она запущена.

Если запущенной «чужой» программе, будь это даже АП, ничего не передавать в виде параметров, то для достижения своих агрессивных планов, необходимо искать ещё одну ошибку в системе для достижения более трудной задачи – нарушить систему защиты файлов.

Агрессивные действия

Однако в существующих системах, если АП запущена, то в силу обсуждённого дефекта в дизайне ОС, она получает доступ к файлам пользователя и может выполнять большой набор агрессивных действий.

Можно разослать саму себя другим адресатам, пользуясь адресной книгой пользователя.
Можно как угодно модифицировать файлы пользователя.
Можно передать информацию пользователя по любому адресу.
Можно исполнять приказы автора АП, считываемые через сеть.
Можно так модифицировать исполняемые файлы пользователя, чтобы при каждом их запуске, работала АП, и там самым было бы обеспечено её постоянное присутствие и т.д.

Заключительные замечания

Две причины порождают проблемы в связи с АП:

-наличие ошибок в разработке и реализации системных программ, которые позволяют АП проникнуть на машину пользователя и быть запущенной

-организация файлов, открывающая доступ к файлам пользователя, любой запущенной программе

Первую причину можно существенно ослабить, с помощью технологии, описанной в разделе *Реализация*. Вторая причина может быть полностью исключена, с помощью организации системы файлов реализованной в Эльбрусе 1, 2 и описанной ниже.

Система файлов Эльбрус

Описываемая система файлов была реализована в Эльбрусе 1, 2. она не потребовала дополнительной аппаратной поддержки по сравнению с описанной в предыдущих разделах.

В Э3-Е2К по соображениям совместимости используются существующие ОС. Не представляет большой проблемы реализовать на Э3-Е2К ОС, подобную ОС Эльбрус 1,2. Проблемы её реализации на Э3-Е2К будет обсуждён позже.

Описываемая система полностью основана и реализует подход, описанный в разделе БАЗОВЫЕ ИДЕИ, адаптированный к системе файлов.

Общая организация системы файлов Эльбрус

В системе существуют два типа объектов: файлы и справочники. Каждый файл имеет заголовок. Все заголовки файлов собраны в отдельный файл заголовков (ФЗ), недоступный непосредственно пользователям. Все справочники собраны в другой файл справочников (ФС), так же недоступный пользователям непосредственно. Все эти файлы находятся в контексте ОС и не включаются ни в какой пользовательский контекст.

Ссылка

Как известно, в реализации БАЗОВЫХ ИДЕЙ основополагающую роль играет ссылка.

Прежде всего, следует обсудить два вопроса:

- на что ссылка может указывать и
- где она сама может размещаться.

На что указывает ссылка

Можно было бы не вводить ни каких ограничений на область, на которую указывает ссылка. В этом варианте каждая ссылка в качестве адреса должна была содержать идентификацию файла, смещение внутри файла до начала области и размер области.

Однако в этом нет большой необходимости.

Когда файл открыт и идёт с ним работа в памяти, то, разумеется, обращение происходит к его различным частям. Однако здесь используются ссылки из памяти на открытый файл.

Сейчас же мы должны рассмотреть систему файлов в процессе хранения данных, до начала их обработки. В этом случае вполне достаточно иметь ссылки на файл как целое. Этого достаточно, чтобы указать, какой файл необходимо открыть для работы или как реструктурировать систему файлов в целом. Введение такого ограничения, не накладывая каких бы то ни было ограничений на программирование, сократит длину ссылки (можно опустить смещение и размер области).

Ссылка из системы файлов, разумеется, не может указывать на объекты в оперативной памяти. Это ограничение вполне понятно. Не может быть ссылки на объект с более коротким временем жизни.

Где ссылка располагается

Универсальным является возможность размещать ссылку в произвольном месте файла.

Однако, как будет видно из дальнейших обсуждений, крайне важно иметь все ссылки сосредоточенными в обозримой области. Кроме того, размещение ссылок в теле файла потребовало бы для их надёжной защиты применения техники, подобной тегам в памяти, что крайне не желательно.

Но самое главное, это совсем не является необходимым. Размещение ссылок только в заголовке файла не приводит к заметной потере эффективности. Если все ссылки собраны в заголовке, то это не приводит к каким либо ограничениям в программировании, но для использования таких ссылок из произвольного места потребуется одно дополнительное обращение в память (чтобы считать указатель на файл по индексу из СВС открытого заголовка, см. ниже в данном разделе).

Это вполне допустимо, так как ссылка из файла на другой файл типично используется для открытия последнего, что само по себе требует значительного времени работы машины.

В результате, в системе Эльбрус предусмотрены ссылки из одного файла на другой как целое.

Пользовательские программы, разумеется, не имеют прямого доступа к ссылкам, для этого существуют системные программы. Но с помощью этих системных программ, пользователь может выполнять любые разрешённые операции со ссылками.

Все ссылки из данного файла сосредоточены в его заголовке. Для этого заголовок файла может содержать массив ссылок - СЕГМЕНТ ВНЕШНИХ ССЫЛОК (СВС).
Если данный файл содержит программу, то из этой программы доступны все файлы, ссылки на которые помещены в СВС этого файла.
Если кто-либо имеет ссылку на файл, то ему доступны и все файлы, ссылки на которые помещены в СВС этого файла.

Кроме того ссылки содержатся в справочниках. Ссылка может адресовать либо весь справочник, либо его отдельный вход.

В результате, все ссылки сосредоточены в двух файлах – ФЗ и ФС.

Сама ссылка содержит права доступа и адрес файла или справочника (признак файл/справочник и смещение по ФЗ до заголовка файла или смещение по ФС до справочника, на который указывает ссылка).

Так как входы справочника могут указывать вновь на справочник, можно использовать слоговые имена.

Права доступа могут разрешать модифицировать или исполнять файл, на который указывает ссылка. Ссылка на программу, в связи с проблемой контекста, будет обсуждена более подробно в следующем разделе.

В отличие от существующих систем, **в Эльбрусе, при обращении пользовательской программы к файлу с использованием ссылки, система не должна проверять специальных санкций от владельца файла для разрешения доступа. Проверяются только права доступа, указанные в ссылке. Если пользовательская программа имеет эту ссылку в своём контексте, то тем самым доступ ей гарантирован.** Это следствие самого фундаментального подхода, общего для виртуальной памяти и для файлов, так как все ссылки находятся под контролем системы, и никакой пользователь не может создать или произвольно модифицировать ссылку.

Можно было бы ввести ещё два полезных права доступа. Это полная константа, обсуждённая ранее и доступ до объекта (файла или справочника) без права копирования, использованной для этого доступа, ссылки. Эти права не были реализованы в Эльбрусе.

Сами заголовки ФЗ и ФС, как и любых других файлов, находятся в ФЗ. Физический адрес заголовка ФЗ и ссылки на ФЗ и ФС после раскрутки ОС, находятся в памяти в контексте ОС.

Контекст программы исполняемого файла

В соответствии с изложенными выше БАЗОВЫМИ ИДЕЯМИ, всякая исполняемая программа ограничена контекстом, предусмотренным её разработчиками. Введённые в системе Эльбрус ссылки в виртуальной памяти и в системе файлов как раз и позволяют, достаточно точно и удобно для программиста, ввести такой контекст для любой работающей программы.

Контекст программы

Контекст работающей программы достаточно подробно был рассмотрен в предыдущих разделах, где было отмечено, что глобальный контекст любой запускаемой процедуры, как свою часть, включает контекст файлов.

Глобальный контекст работающей программы в оперативной памяти, вообще говоря, динамическое понятие. Он создаётся в памяти во время работы. Более того, различные запуски одной и той же программы (например, в разных виртуальных пространствах или в качестве различных поколений рекурсивного запуска) имеют глобальные контексты, расположенные в различных областях виртуальной памяти. Поэтому невозможно статически, до запуска программы связать код с глобальным контекстом, его просто не существует в это время.

Однако с частью глобального контекста, представляющим файлы, дело обстоит проще. Так как эта часть обычно создаётся автором программы – файла до её исполнения, её жёстко, статически можно связать с кодом, т.е. с самим файлом. Это не накладывает ни каких ограничений на программы и на их контекст. Этот контекст при необходимости может меняться. Например, если в контексте находятся справочники, содержание которых может меняться программами.

Это исключает возможность запускать одну и ту же программу с различным внешним окружением в простейшем варианте запуска. Вариант запуска программы с различным внешним контекстом практически не нужен совсем, но в случае необходимости, он может

быть легко реализован введением дополнительной системной программы. Однако это можно делать, только если в контексте запускающего есть доступ (ссылки) к файлу кода (с правом читать, а не только исполнять) и ко всем файлам составляющим внешний контекст. В противном случае, это будет нарушение защищённости.

В системе Эльбрус весь внешний контекст файлов программного кода находится в СВС файла, в котором этот код расположен. Кода запускается программа, открывается исполняемый файл, его заголовок размещается в памяти вместе с СВС и в глобальном контексте программы в памяти размещается ссылка на массив внешних ссылок. Таким образом, внешний контекст файлов становится частью общего глобального контекста программы. Поэтому, ссылка на исполняемый файл не должна содержать информации о контексте, так как контекст включается самой программой.

Организация внешнего контекста является одним из наиболее важных отличий системы Эльбрус от существующих ОС. По существу, каждой отдельной процедуре предписывается свой индивидуальный контекст файлов.

Про существующие системы можно сказать, что у всех исполняемых файлов, расположенных на данной машине, один и тот же внешний контекст файлов – это корень системы файлов данной машины. В каждом же файле перечислены пользователи, которым разрешён доступ к нему.

В Эльбрусе же наоборот, корня системы не существует вообще (точнее он не доступен пользователям), но в каждом исполняемом файле кода перечислены ссылки, по которым только этот код может обращаться к внешнему окружению файлов.

Существующие системы можно назвать **root centric**, а систему Эльбрус – **file centric**.

Контекст пользователя

Пользователя, сидящего за дисплеем машины или каким либо другим способом, запускающим на счёт машину, можно, также представить как вариант файла кода. Его так же необходимо ограничить контекстом. Однако в отличие от программы, адресация в этом контексте должна быть не с помощью «откомпилированной цифры», наподобие СВС, а с помощью символьных имён, так как их использует программист.

Поэтому контекст пользователя представлен справочником, в котором входным ключом является символьная строка, а выходом ссылка.

Как и СВС программы, так и контекстный справочник пользователя, может ссылаться вновь на справочник и тогда пользователь, для доступа к файлу или справочнику, должен (или может) использовать многослоговое символьное имя.

Общая структура системы файлов ОС Эльбрус

Общая структура системы выглядит следующим образом \\\\\\\\\\(рис. 2)//////////.

В контексте ОС существует справочник пользователей работающих на данной машине. Каждый вход справочника ссылается на документацию данного пользователя (системная информация). В документации пользователя есть ссылка на контекстный справочник пользователя. Этот справочник является корнем дерева справочников данного пользователя содержащего все его файлы. Однако этот корень не доступен, вообще говоря, его исполняемым программам. Как уже описывалось выше, контекст каждого исполняемого файла содержится в его СВС. Ссылки из этого СВС могут адресовать как файлы того же пользователя, так и любые другие файлы в системе.

Таким образом, файловая система Эльбрус представляет собой не традиционную древовидную, а сетевую структуру.

Следствием этого факта является необходимость запускать периодически программу сбора мусора. Эта программа работает только с заголовками файлов и со справочниками, т.е. с содержимым двух системных файлов ФЗ и ФС, что позволяет выполнять эту работу достаточно эффективно.

Файлы в языках программирования

Проблемы, обсуждаемые в данном разделе, в одинаковой мере относятся и языкам программирования и языкам, которые призваны оперировать с объектами на уровне операционной системы (скрипты, shell и т.д.). В принципе нет особой необходимости вводить для этой цели специальные языки. Наличие таких языков скорее свидетельствует о недостатках в основных языках программирования.

В Эльбрусе 1, 2 основной язык программирования Эль 76 быль спроектирован таким образом, что он с успехом заменял языки операционных систем.

Как уже отмечалось выше, для эффективного использования разработанной системы файлов, языки программирования необходимо расширить.

Во-первых, следует ввести указатели на файлы и справочники. С помощью этих указателей файлы можно передавать в качестве параметров в запускаемую процедуру.

Другим необходимым расширением является объявление и определение внешних имён.

Все внешние имена (однослоговые идентификаторы) используемые в программе должны быть в ней объявлены. Каждому такому внешнему имени после трансляции будет соответствовать вход в СВС. Если этот вход СВС содержит ссылку на справочник, то в программе это внешнее имя может использоваться в качестве первого слога многослогового имени.

Определение этого имени (объект, на который указывает ссылка в данном входе СВС) происходит в контексте, который передаётся транслятору в качестве параметра при трансляции. Типично это контекст пользователя - автора программы.

Чтобы не открывать контекст пользователя транслятору, можно после трансляции использовать системную процедуру для заполнения СВС. Только этой процедуре и будет необходим контекст пользователя.

Если в момент трансляции указываемого объекта ещё не существует, в системе предусмотрено отложенное связывание. Разумеется, во время исполнения объект должен существовать.

Новые проблемы

В данном разделе будут приведены некоторые новые проблемы, которые отсутствовали в существующих системах. Будут обсуждены методы их решения.

Передача

В существующих системах не представляет проблемы обратиться к файлу другого пользователя. Общий корень, доступный каждому, позволяет именовать файл любого другого пользователя. Чтобы выполнить доступ к чужому файлу достаточно, чтобы его владелец разрешил это сделать.

В новой системе контексты пользователей, вообще говоря, разделены, хотя, система допускает любую, желаемую для пользователей, степень пересечения. Встаёт вопрос, как может один пользователь обеспечить другому доступ к своему файлу, если до этого эти два пользователя не имели ни какого информационного пересечения. Эта новая проблема для этой системы.

Решение её достаточно очевидно.

Один пользователь может передать ссылку на свой файл другому через локальную почтовую службу. Это не обычная электронная почта, а специальная система, которая работает следующим образом.

Любой пользователь может попросить систему передать ссылку другому пользователю, указав его символьное имя. Пользователь - получатель также должен обратиться к системе, с просьбой поместить ссылку, полученную от поименованного пользователя – отправителя, в определённое место своего информационного пространства.

Хотя эта система проста в использовании, пользоваться ею необходимо крайне редко, при установлении общего пространства между пользователями, между которыми его не было. Если общее пространство уже существует, то далее ссылки можно передавать через него.

Отъём

Аналогично, новая для описываемой системы проблема связана с обеспечением возможности отъёма права доступа к файлу у другого пользователя, которому ссылка на этот файл была передана ранее.

Существует несколько способов решения этой проблемы.

Хозяин ссылки может попросить систему удалить указанную ссылку из контекста пользователя, которому она была передана. Эту работу система может выполнить, проведя соответствующий анализ части файлов ФЗ и ФС, так как в каждом файле есть имя пользователя. Можно попросить удалить эту ссылку у всех пользователей кроме хозяина и т.д.

Эта работа не столь трудоёмкая, не сложнее однократного просмотра всех заголовков.

Можно использовать другую возможность. Владелец файла может переместить его в другое место, перенацелив на него все свои ссылки, а на старое место поместить «засаду».

В этом контексте полезно было бы ввести в указатель файлов, среди других прав, упомянутое выше право копирования.

Если, например, ссылка на файл без права копирования передана программе другого пользователя, то он может свободно ею пользоваться в данной программе, передавать её другим программам в качестве параметра, но ни он, ни эти другие программы не могут записать её ни в какой справочник или СВС. Эта ссылка автоматически прекратит своё существование в конце задачи, в которой она использовалась и хозяину не придётся её отнимать.

Список ссылок

В существующих системах имя файла или список имён файлов может храниться среди данных пользователя. В новой системе, ввиду обсуждённого выше ограничения, этого делать нельзя. Ссылка это специальный тип данных и его можно хранить только в справочниках или в СВС.

Справедливости ради следует отметить, что хранение имён файлов в программе – это не лучший способ программирования.

Тот стиль, который навязывает защищённая система, гораздо более надёжен.

Работа в памяти

С технической точки зрения работа пользовательских программ с системой файлов организована с помощью тех же методов, которые легли в основу всей системы – это определяемые типы данных.

Структуры заголовка файлов, справочника, ссылки, их представление на внешнем носителе и в памяти, когда они бывают считаны для оперативной работы, всё это новые типы, определённые программно. Их внутренние структуры, различные поля доступны для чтения и модификации только в контексте процедур, специально для этого написанных и составляющих поддержку этих типов.

Описываемая система файлов была реализована на Эльбрусе 1, 2. Если бы она была реализована на Э3-Е2К, то была бы использована стандартная техника определяемых типов.

Пользователь имел бы в своём контексте только ОД указывающий на считанные в память заголовок или справочник и т.д. в качестве приватных данных, не доступных для прямого использования пользователем.

В Эльбрусе 1, 2 вместо ОД, был введён специальный тегированный тип 64 разрядной величины. По существу это был специальный дескриптор, содержащий виртуальный адрес и размер области, где содержалась более подробная информация, описывающая объект.

Пользовательские программы могли его использовать только для передачи в качестве параметров процедурам обработки.

Этот тегированный дескриптор или указатель (УК) мог указывать на:

- заголовок открытого файла
- справочник
- файл (закрытый)
- один вход справочника
- один вход СВС

С помощью этого УК можно было выполнять различные операции (использовавшие УК в качестве параметра или вырабатывавшие его), такие как:

- открыть/закрыть файл
- создать/уничтожить файл
- создать/уничтожить справочник
- записать ссылку во вход справочника/СВС
- передвинуть УК (навигация) через справочник/СВС
- «пройти» от данного УК через многослоговое имя.

Возможные расширения в межмашинную область

Всё изложенное выше было реализовано в машинах серии Эльбрус и прошло серьёзную проверку в процессе широкой эксплуатации на многих системах пользователей.

В данном же разделе будет обсуждено возможное расширение предложенных принципов, по существу, на область распределённых ОС.

Разумеется, сама возможность реализации данных расширений должна подвергнуться тщательному изучению. Однако первоначальное рассмотрение позволяет надеется на положительный результат.

Ссылка

Так как в основе разработанного подхода лежит реализация защищённой ссылки, прежде всего здесь будет рассмотрена возможность её реализации в сети.

Попробуем построить защищённую ссылку на файл расположенный в другой машине.

Вспоминая принципы взаимного доверия (или взаимного недоверия) изложенные в разделе *Единица компиляции*, при построении ссылки на файл в другой машине, следует обеспечить её полную защищённость в соответствии с изложенными принципами, несмотря на полное недоверие между всеми системами разных машин.

Рассматривается следующая организация такого рода ссылки.

Ссылка выдаётся машиной А на свой файл по просьбе или согласованно с машиной В. Куда конкретно в машине В будет размещена эта ссылка, машину А не интересует.

На каждую ссылку выданную другой машине (например, В) машина А сохраняет «обратную ссылку» - или разрешение машине В обращаться к данному своему файлу.

При каждом обращении машины В к файлу машины А, А проверяет разрешение.

Если ссылка была выдана с правом дальнейшего распространения, то машина В, при передаче ссылки на файл А машине С, обращается к машине А, которая установит (автоматически) «обратную ссылку» о файле А на машину С.

Если ссылка указывала на справочник и имела соответствующий признак, то машина В может выполнить выборку входа из справочника и получить соответствующую ссылку автоматически (разумеется с установкой «обратной ссылки»).

Предлагаемая схема обладает достаточной степенью защищённости. Действительно, как и внутри одной машины, передавая файл другому пользователю, хозяин подвергает опасности только переданный файл и только в объёме выданных прав доступа.

Рассмотрим теперь некоторые проблемы, связанные с введением такого рода ссылки и их возможные решения.

Public/private

Если машина А решила позволить открытый доступ к какому-либо своему файлу с учётом прав доступа (например, публично только читать), то вместо «обратной ссылки» ставится признак **public**. Такая оптимизация позволяет не тревожиться, что в случае публичных файлов переполнится список «обратных ссылок». Кроме того, если ссылка была выдана с признаком **public**, то упрощается её дальнейшая передача, не нужно обращаться к машине – хозяину для учёта «обратной ссылки».

Перевод владельцем файла состояние файла **public/private** и обратно, так же, не вызывает трудностей с точки зрения защищённости. Следует просто изменить запись в списке «обратных ссылок» ни к кому, при этом, не обращаясь.

Мусорщик

Если машина А – владелец файла обнаружила, что этот файл превратился в мусор с точки зрения внутреннего использования, но на него существует внешняя ссылка из машины В, то тут можно просматривать две возможности.

Во-первых, во всех случаях необходимо спросить у машины В (разумеется, у системы, а не у человека), нужен ли он ей, т.е. не является ли эта ссылка «мусорной» и в машине В.

Ответ может потребовать опроса других машиной В. Однако это не процесс обычного мусорщика, так как существуют обратные ссылки. Но и этот процесс необходимо правильно организовать. Это вполне можно сделать, но здесь не будут обсуждаться его детали.

Во-вторых, в случае если файл кому-нибудь оказался нужен снаружи, можно либо сохранить его до освобождения, либо потребовать переместить его на другую машину, туда, где он необходим.

Если файл оказался мусором у машины – владельца и ссылка на него предварительно была выдана наружу, простой просмотр вариантов поведения других машин показывает, что они не могут причинить ни какого вреда владельцу.

Контекст

Контекст программ

При наличии межмашинных ссылок глобальный контекст программы может включать ссылки на файл, расположенный на другой машине.

Если, например, пользователь, работающий на машине А, перешлёт свой исполняемый файл включая СВС, на машину В, то ссылки из его СВС будут указывать на файлы в машине А. Программу можно будет исполнять и она во время работы будет обращаться за файлами из своего глобального контекста, через свой СВС и через прямые межмашинные ссылки, к файлам на машине А.

Можно установить прямую межмашинную ссылку на публично доступный файл, расположенный на другой машине.

При запуске программы на одной машине, ей может быть передан в качестве параметров файл с другой машины.

Все эти примеры иллюстрируют, что с точки зрения функциональности, границы между машинами в значительной мере оказываются стёртыми без каких бы то не было жертв для защищённости.

Контекст пользователя

Контекст пользователя в такой системе только расширится. Можно иметь прямые ссылки на файлы из других машин.

Кроме того, Интернет пространство можно считать расширением контекста пользователя.

Общие соображения

Введение межмашинных ссылок не предполагает решение таких вопросов в управлении сетью, как автоматическое перемещение информации в сети для достижения максимальной эффективности.

Этот шаг связан только с обеспечением функциональной возможности работать в распределённой системе и защищённости.

Пользователь сам определяет место создания и возможное перемещение информации.

И, конечно же, такими межмашинными ссылками, по всей видимости, разумно будет пользоваться с определённой осторожностью. Если, например, необходимо просто переслать файл из одной машины в другую, то не следует это делать путём передачи ссылки и затем считывания этого файла через неё. Файл, как и раньше, следует просто переслать.

Однако рассматриваемая система должна облегчить создание более автоматизированных надстроек в духе системы Grid, так как она обеспечивает решение таких важных для сети вопросов, как функциональная работоспособность независимо от положения информации в сети и строгая защищённость.

Анализ

Надёжность

Разработанная система обладает высокой надёжностью.

Это является следствием следующих обстоятельств.

Во-первых, работающая программа имеет доступ ко всем нужным и **только** к нужным данным. В этой системе право доступа передаются вместе с данными. Разработчик программы, специфицируя её внешнее окружение, тем самым обеспечивает ей доступ к этим и **только** этим данным. Запускающая программа передаёт ей параметры и вместе с ними права доступа к ним и **только** к ним.

Таким образом, в отличие от существующих систем, объём доступной программе информации строго минимизирован.

Во-вторых, надёжность этих ограничений обеспечивается правильным функционированием весьма ограниченного объёма программ реализующих работу ссылки и контекста.

В-третьих, реализация самой ОС в описанной защищённой системе с высокими отладочными возможностями ещё больше повышает уверенность в том, что в этих программах удастся значительно сократить количество ошибок.

Приведённые выше при обсуждении традиционных систем примеры (см. раздел *Традиционные системы файлов*) не вызывают в Эльбрусе ни каких проблем.

В первом примере пользователь ПВ, создавая программу ФВ, с помощью ссылок в СВС снабдил её необходимым контекстом, включающим файл ФВ1.

Пользователь ПА, имея ссылку на ФВ с правом исполнения, запускает её, передавая в качестве параметра ссылку на свой файл ФА. Эту ссылку рационально передать без права копирования. В этом случае она не попадёт ни в какие другие файлы, а будет уничтожена в конце задачи вместе с виртуальной памятью.

Пользователь ПА, не имея доступа ни к коду программы ФВ ни к файлу ФВ1, не сможет причинить какого-либо вреда каким-либо файлам пользователя ПВ, ни по ошибке, ни по злому умыслу.

Сам пользователь ПВ не имеет доступа к файлу ФА, этот доступ имеет только его программа. Разумеется, он может модифицировать её код, либо по ошибке, либо по злому умыслу, что, конечно, создаст проблемы для ПА. Однако это неизбежный минимум. Это должно классифицироваться не как нарушение защищённости, а скорее как неправильное выполнение объявленных функций без нарушения защищённости.

Нет проблем и во втором примере.

Неблагонадёжная программа, запущенная пользователем П, не имеет доступа ни к каким его файлам, кроме тех, которые он сам передал этой программе в качестве параметров.

Простота использования

Рассматриваемая система не требует от программиста **ни каких** усилий для обеспечения защищённости. В руководствах Эльбруса нет ни одного слова о защите и защищённости, несмотря на то, что система обеспечивает предельно совершенную защиту.

Эффективность

Обсуждаемая часть системы файлов касается в основном механизма адресации и поиска нужных файлов. В этой части рассматриваемая система не уступает в эффективности существующим.

В отличие от них, где поиск всегда ведётся по слоговому имени через систему справочников, здесь в большинстве случаев файл находится по прямой физической ссылке, что значительно более эффективно.

Стиль программирования

С точки зрения языка или скорее интерфейса с ОС, основными изменениями в системе Эльбрус являются введение типа данных – ссылки на файл вместо использования для этой цели символьной строки, а так же объявление и определение внешних имён программы (внешний контекст).

Оба эти изменения полностью соответствуют правильному стилю в построении языков.

Их введение делает программы более читабельными, существенно упрощает отладку, облегчает эксплуатацию программ.

ВИРУСЫ И АГРЕССИВНЫЕ ПРОГРАММЫ

В предыдущих разделах обсуждались причины, которые позволяют вирусам и другим агрессивным программам добиваться успеха в существующих системах.

В данном разделе будет обсуждён вопрос, как технология Эльбруса помогла улучшить ситуацию в этой области.

Прежде всего, несколько общих замечаний.

Определённая часть системы, обеспечивающей защищённость от последствия ошибок в программах пользователей, сама должна быть достаточно хорошо отлажена. Трудно себе представить такую систему, которая бы успешно работала, вне зависимости от ошибок в ней самой.

Все понимают, что ошибок в программах избежать пока, к сожалению не удастся. Отсюда, однако, не следует делать вывода, что перед проблемой обеспечения защищённости все программы находятся в одинаково безнадёжном положении.

Правильной целью при создании защищённой системы может быть следующее положение.

Система обеспечивает защищённость, несмотря на любые ошибки или злонамеренные действия в пользовательских программах. Это свойство должно базироваться на требовании высокой отлаженности как можно **меньшей** её части.

Представляется, что описываемая система удовлетворяет этим требованиям.

Тем ядром, которое должно быть хорошо отлажено, являются программы, реализующие понятие ссылки и контекста, как в памяти, так и в системе файлов, и работу с ними. Это не такой уж большой объём программ.

В этом предположении рассмотрим работу системы Эльбрус.

Предположим, что агрессивная программа оказалась запущенной в системе Эльбрус. Несмотря на то, как это произошло, легально или нет, не будем считать этот факт успехом АП, так как сам по себе он не привёл к нарушению защищённости.

При правильной работе механизма ссылок и контекста, этой программе не будет передан какой-либо локальный контекст.

Эта программа будет иметь доступ лишь к своим параметрам, если программа была запущена легально, и к системным процедурам.

Задача АП, нанести вред и/или размножиться.

АП не может без использования ссылок добраться до чужих ячеек памяти или до файлов, и тем самым нанести вред или размножиться. Она может попытаться это выполнить либо через параметры, либо через системные вызовы.

Параметры

Параметры могут содержать ссылку в памяти и в файлах.

Если это ссылка в памяти или файлах на данные, то даже если эта ссылка разрешает модификацию, это может классифицироваться лишь как ошибка, но не как злонамеренные действия. Будут испорчены результаты работы данной программы, которые в дальнейшем будут использоваться внутренними программами, что, скорее всего, приведёт рано или поздно к аварийной сигнализации. Не видно, как в руки АП может попасть ссылка, разрешающая доступ к закрытой от неё информации. И уж, во всяком случае, отсутствуют возможности размножения.

Если это ссылка в памяти или файлах на исполняемую функцию, то вредный эффект может заключаться в том, что эта функция поставит в качестве своего значения опасную ссылку. Вероятность этого исчезающе мала, так как этот параметр был явно передан «сомнительной» программе.

Более подробно следует рассмотреть случай, когда в качестве параметра передана ссылка на исполняемый файл с правом его модификации.

В этом случае АП может его инфицировать, т.е. модифицировать код программы так, что она сама станет вариантом АП с возможностью дальнейшего размножения.

Во-первых, ссылка на исполняемые файлы с правом их модификации представляет собой крайне редкий случай. Все файлы пользователя делятся на файлы данных и на программы. В обычной жизни практически ни когда не требуется передавать кому-либо ссылки на программы с правом их модификации. Только владелец программного кода имеет на него ссылку со всеми правами.

Но даже если это произошло, дальнейшее размножение так же крайне мало вероятно. Для этого в контексте этой инфицированной программы должны находиться подобные же крайне редкие ссылки на исполняемые файлы с правом их модификации. И так далее на каждом шагу.

Всё это делает вероятность реального размножения близкой к нулю. При таких вероятностях труд создателей вирусов становится совершенно не эффективным и ни кто не станет им заниматься.

Системные вызовы

Обращаясь к системе, АП может надеяться, что, правильно подобрав параметры и используя ошибки в системных программах, удастся получить, в качестве возвращаемого значения, нужную ссылку.

Вероятность этого так же крайне мала, так как, во-первых, вся система, написанная в системе Эльбрус, имеет значительно более высокую степень отлаженности.

Во-вторых, каждая системная процедура работает в резко ограниченном контексте, с малой вероятностью содержащем «полезные» для АП ссылки, так что при любых ошибках в ней такая процедура не сможет быть «полезной» для АП.

Конечно, и в этой системе существует зависимость от фактических ошибок в системе, и нет 100% гарантии защиты от АП.

Однако для того, чтобы работа по созданию вирусов стала бессмысленной, и ею прекратили бы заниматься, нет строгой необходимости свести эту вероятность к нулю. Достаточно, чтобы она был значительно уменьшена.

В системе Эльбрус драматическое сокращение вероятности достигается, по крайней мере, по двум причинам.

Во-первых, ввиду большей отлаженности системы, «вирусологу» гораздо труднее добиться запуска АП.

Во-вторых, ликвидируется огромная «дыра», когда считанной их сети программе обеспечивается доступ ко всем файлам пользователя.

Тем самым, если даже АП оказалась запущенной, на что ушли все «творческие» силы создателя вирусов, то теперь в системе Эльбрус ему ещё надо, используя ещё какие то ошибки системы, добиться доступа к чувствительным файлам, что при более отлаженной системе ещё более трудно сделать.

Безусловно, можно говорить о драматическом сокращении вероятности «успеха» для АП и, тем самым, об успешной борьбе с вирусами.

ПРОБЛЕМА СОВМЕСТИМОСТИ И ВНЕДРЕНИЯ

Описанная технология защищённой системы в аппаратуре, в операционной системе, в системах программирования и в остальных системных программах была полностью реализована и прошла тщательную проверку практикой использования в двух поколениях вычислительных систем (Эльбрус 1 и 2) широко распространённых в Стране.

Одним из наиболее типичных применений этих машин были системы реального масштаба времени, так что при их использовании была продемонстрирована высокая эффективность, превосходящая по уровню другие конкурентные системы того времени в стране.

В проекте Э3-Е2К была проведена значительная модернизация архитектуры, повышающая её эффективность в применении к современным языкам и ОС.

Эта технология продемонстрировала высокие качества защищённости, позволяющие на порядок упростить и ускорить отладку программ, значительно сократив тем самым время их разработки (time to market) повысить их качество, значительно снизив вероятность появления ошибок в поставленных пользователям программных системах.

Наконец, эта система позволяет надеяться на полное решение такой актуальной для настоящего развития всей компьютерной индустрии проблемы, как борьба с вирусами и другими агрессивными программами и упростить создание распределённых информационных систем.

Несмотря на такие значительные преимущества, надежды на использование этой технологии в распространённых микропроцессорах и системных программах исчезающе мала, по крайней мере, в ближайшее время. И причины здесь носят далеко не технический характер.

Однако представляется разумным рассмотреть те шаги, которые могли бы привести к её использованию.

Дальнейшие обсуждения будут вестись в соответствии со следующей схемой.

Разобьём технологию Эльбрус на несколько последовательных (иногда параллельных) шагов. Каждый последующий шаг улучшает ситуацию, но требует больше работы.

Прежде всего, будут рассмотрены необходимые минимальные расширения языков и интерфейса ОС, которые сами по себе обеспечат значительный шаг в направлении улучшения защищённости существующих систем, но, кроме того, обеспечат возможность вводить последующие усовершенствования.

Кроме этих изменений, для реализации следующих шагов ни каких других изменений в языки или интерфейс ОС вводить не потребуется.

Следует отметить, что в Эльбрусе был реализован полный вариант технологии (за исключением межмашинных ссылок), поэтому возможные трудности, которые могут возникнуть на промежуточных шагах, следует изучать особо.

Шаг 1. Базовые усовершенствования

Этот шаг связан с расширением языков путём введения нового типа данных «имя файла» и введения описания и определения внешних имён файлов в программе.

Выполнять этот шаг можно в двух вариантах. Второй вариант может рассматриваться как продолжение первого.

Первый вариант

1. В языки вводится тип данных «имя файла», который можно передавать в качестве параметра, в том числе в другие независимо откомпилированные программы. Для ОС, по всей видимости, это означает введение и указателя на такой тип данных. Для файлов, переданных в программу с использованием указателей, не проверяются права доступа, они присутствуют в указателе. Система проверяет права доступа при формировании указателя. Разумеется, указатели должны храниться в системной памяти.

Это действие не нарушает совместимости. Во всех операциях, кроме типа «имя файла», можно продолжать пользоваться и строками.

2. Программы, считанные из сети, метятся специальным признаком.

Это действие, так же не нарушает совместимости.

3. Помеченным программам запрещается обращаться к файлам с использованием в качестве имени строки. Они должны использовать для этой цели указатели. Иными словами, считанные из сети программы могут пользоваться только файлами, переданными им в качества параметров. Это как раз то, что и требовалось достичь.

Это действие так же не нарушает совместимости для всех внутренних программ. Но оно нарушает совместимость для программ, считанных из сети. А это значит, что многие системные программы, такие как почта или программы обслуживающие Интернет, придётся откорректировать.

Однако уже этот вариант позволяет «заштопать» ту огромную «дыру», которая в большой степени повинна в беспомощности современных систем перед угрозой вирусов.

Все дальнейшие шаги позволяют избавить систему от множества ошибок, не в меньшей степени повинных во всех бедах существующих систем.

Второй вариант

1. В исполняемых программах вводится описание и определение внешних имён файлов.

Совместимый шаг.

2. В исполняемых программах запрещается пользоваться строками в качестве имён файлов. Можно пользоваться либо предварительно описанными внешними именами, либо параметрами, переданными с помощью указателей.

Несовместимый шаг. Вводит хорошую модульность при работе программ пользователей с файлами.

Шаг 2. Модернизация аппаратуры

Вводится описанная выше архитектура машины

Этот шаг призван ввести высокий уровень защищённости при работе программ пользователей.

Он не вводит новой несовместимости на уровне языков и ОС. Но на уровне двоичных кодов вводится новая архитектура.

В проекте Э3-Е2К разработана технология двоичной компиляции, которая решает эту проблему и обеспечивает возможность работать как со старыми двоичными кодами, так и в защищённом режиме. Программы, отлаженные в защищённом режиме, могут затем исполняться на Э3-Е2К или на других существующих машинах в совместимых двоичных кодах (разумеется, после перетрансляции).

Этот шаг повышает защищённость пользовательских программ.

Шаг 3. Реализация ОС в защищённом режиме

ОС реализуется в защищённом режиме.

Этот шаг не вводит новой несовместимости.

Именно он должен значительно сократить число ошибок в системе и, тем самым, значительно увеличить защищённость системы в целом.

ЗАКЛЮЧЕНИЕ

Неизвестно, существует ли другой способ борьбы с «чумой» современной компьютерной индустрии, но рассмотренная здесь технология решает эту проблему, безусловно, концептуально наиболее простыми и ясными методами, а значит наиболее надёжно.

Эта технология эффективна и хорошо проверена в эксплуатации.

Было бы крайне печально, если этот опыт будет забыт и люди по прежнему будут жить в ожидании очередного вируса, чтобы изучить его и найти способ его нейтрализации, уже после того, как он успеет натворить много бед, понимая, что это единственный способ защиты и «теоретически» доказав, что таким бедам нет конца.

ИСТОРИЯ

Достаточно полный обзор систем на базе использования дескрипторов и идеи capability содержится в книге Levi [1].

Пионерами этих идей в аппаратной части являются Роберт Бартон и Джон Айлиф. В 50ые годы они работали вместе в одной из фирм, где, по всей видимости, обсуждали эти идеи. Затем Бартон продолжил работу на фирме Барроус и руководил разработкой коммерческих машин B5000 [4] и B5500 (1961), а Айлиф в Лондонском Университете выполнил исследовательский проект Basic Language Machine (1964) [13] и построил экспериментальную машину на его основе. Он, так же был связан с созданием машины Университета Rice (1959) [10].

Машина B5500 имела весьма хороший коммерческий успех.

В машине B5000 ссылка получила название ДЕСКРИПТОР, а Айлиф называл её CODEWORD.

В 1966 году Денис и Ван Горн в работе [3] ввели понятие **capability** и разработали некоторые принципы работы таких систем. Они же затем использовали эти принципы в экспериментальной ОС на машине PDP-1.

Это было самое начало. Все эти машины поддерживали в аппаратуре ссылку, как базовый элемент доступа к данным и аппаратура контролировала правильность работы с ней. Для этого все ссылки должны быть отличимы аппаратурой от остальных данных.

Уже на этой ранней стадии определилось два возможных варианта реализации этой идеи.

Первый вариант, получивший название **capability list approach** или **C-list,** требовал размещения все ссылок в специальных сегментах памяти. Будем его называть вариантом *Списка Ссылок* (СпС).

Второй вариант не накладывал ограничения на размещения ссылок среди прочих данных, но предусматривал пометку каждой ссылки в памяти специальным битом (битами) - ТЕГом, который использовался только для этого и не мог прямо обрабатываться программами пользователя.

Хотя машина B5000 имела теги, но основу её работы составлял подход СпС. Машины Айлифа так же имели смешанный подход, но в основе так же лежал СпС подход.

Помещение ссылок в отдельный сегмент для системы памяти сильно ограничивало программирование и по существу приводило к неудобствам и потере эффективности.

Этот подход вполне разумен для системы файлов, где эффективность зависит больше от сокращения числа обменов с внешними устройствами, чем от ускорения операций в памяти. В результате, в системе файлов при использовании СпС подхода можно обеспечить пользователю столь же комфортабельную ситуацию, как и в варианте тегов.

В результате опыта, полученного при использовании B5000 и B5500, авторы отмечали несовершенство СпС подхода и недопустимость ограничивать размещение ссылок только в специальных сегментах.

В последующих машинах (B6500 и т.д.) было разрешено размещать ссылки среди скалярных данных. Однако вместе с водой был выплеснут и ребёнок. Пользовательским программам было позволено свободно модифицировать ссылки, включая разряды тегов, подобно данным, среди которых эти ссылки теперь находились. В результате аппаратура не обеспечивала элементарную защиту памяти. Всё это ложилось на плечи компилятора, который, конечно, не мог её обеспечить. В этих машинах защита памяти попросту отсутствовала.

Несмотря на опыт B5000, последующие разработчики продолжали использовать СпС подход.

Этот же подход использовался при разработке некоторых ОС, что имело гораздо больший смысл.

Вот список машин и ОС, последовавших за упомянутыми выше первыми работами, использовавшими СпС подход.

Машина Magic Number Machine[14]	Университет Чикаго	1967	незавершена
ОС CAL-TSS	Университет Беркли	1968	
Машина System 250[9]	Plessey Corp., UK	1969	коммерческая Специализированная
Машина CAP [8]	Университет Кембридж, UK	1970	
ОС Гидра [7]	Университет Карнеги-Мелон	1971	
ОС StarOS [11]	Университет Карнеги-Мелон	1975	
Микропроцессор iAPX 432 [12]	Интел	1981	коммерческая машина

В большой популярности СпС подхода, кроме опыта первых реализаций, повинна, по всей видимости, и работа Дениса и Ван Горна, упомянутая выше. То, что она была очень популярна в то время и оказала влияние на все последующие разработки, в этом нет сомнения. В ней были обоснованы «преимущества» СпС подхода.

При всей, безусловно, положительной роли, которую сыграла она для утверждения этого направления, она имела ещё один дефект. Авторы попытались как можно больше нагрузить введённое ими понятие **capability.** Здесь было и управление процессами и управление сообщениями. И всё это последующие разработчики пытались вложить в аппаратуру, что, конечно, приводило к неэффективности. В то время как всё это следовало бы реализовать с помощью программно определяемых типов, используя объектно-ориентированные методы программирования.

К сожалению, много было дефектов в этих системах совсем не связанных с реализацией ссылок. Вот мнение Levi, автора упомянутого выше обзора.

"…the performance problems suffered by many early capability systems were often due to peculiarities of the individual design or to hardware poorly matched to the task. There is probably no inherent reason why a capability-based system cannot perform well as a conventional architecture machine."

Несколько слов о двух последних коммерческих машинах системе IBM S/38 [5, 6] и последовавшей за ней AS/400 [2] и микропроцессоре iAPX 432, фирмы Интел. Это наиболее важные для обсуждения работы. Обе эти машины были выпущены позже Эльбруса. Эльбрус 1 заработал в конце 1978 года и передан пользователям в начале 1979 года. IBM S/38 объявлена в 1978 выпущена в 1980 году. Интел 432 выпушен в 1981 году.

Система IBM S/38 – AS/400

Эта система единственная, кроме Эльбруса, использующая теги для реализации защищённости.

Машины эти коммерчески выпускались и выпускаются в настоящее время фирмой IBM. Это, безусловно, коммерчески успешный продукт.

Безусловно, правильно, что система не использует СпС подход, введены теги, позволившие свободно размещать ссылки (указатели) среди скалярных данных.

Однако с технической точки зрения многие решения, реализованные в этих машинах, совсем не бесспорные и требуют комментариев.

Общий анализ

Как можно заключить из анализа этой системы, защищённое программирование не было в числе основных целей при создании системы. И, как результат, защищённое программирование не реализовано в системе.

С точки зрения программиста, работающего на этой системе, по защищённости она ни чем не отличается от других систем.

Попытаемся разобраться, почему в этой системе были использованы теги и, какую роль они выполняют.

Начнём с базового принципа, положенного в основу системы и приведшего в результате к использованию тегов и проследим, к каким результатам привела его реализация.

Таким базовым принципом явилась идея *памяти единого уровня*.

В соответствии с этой идеей и используя возможности технологии по реализации большого виртуального адресного пространства, была сделана очередная попытка объединить понятия оперативной памяти и файлов в единую концепцию.

Выше уже была приведена критика такого подхода. Здесь же можно проследить к каким результатам привела попытка её реализации.

В системе было введено единое виртуальное адресное пространство, в которое были загружены оперативные данные всех задач и файлы всех пользователей. При этом была сделана попытка единообразной работы с этими данными.

Для реализации этой цели необходимо было, по крайней мере, два свойства оперативной памяти распространить на данные, хранящиеся в файлах.

В оперативной памяти указатели размещаются вперемежку с другими данными (первое свойство) и различные виртуальные пространства задач обеспечивают защиту оперативных данных между задачами (второе свойство).

Если попытаться ввести единообразие и позволить в системе файлов использовать целые числа в качестве указателей, то не будет обеспечена защита между задачами.

Теги и были введены как раз для того, чтобы примирить эти два требования.

Однако в последствии (при переходе от S/38 к AS/400) оказалось, что ещё одно свойство, теперь уже принадлежащее системе файлов, требовалось обобществить. Это свойство касается возможности для владельца данных, который передал право их пользования другим, в любой момент отобрать это право.

В системе файлов это делается довольно просто, так, например, в момент открытия файла, система может проверить права доступа того, кто хочет с ним поработать.

В системе с тегами и с указателями, разбросанными по всему огромному единому виртуальному пространству, сделать это практически невозможно. Нельзя же всерьёз рассчитывать на сканирование всего пространства, в поисках указателей на данные, доступ к которым необходимо ограничить.

Для того, чтобы справиться с этой проблемой, были введены два типа объектов и указателей их адресующих. Были введены *постоянные объекты* (читай файлы) и *временные объекты* (читай массивы оперативной памяти).

При *каждом одиночном* обращении к постоянным объектам (при каждой операции LOAD\STORE) с использованием микрокода проверяются права доступа, вместо того, чтобы это делать однажды при открытии файла.

Это, во-первых, приводит к значительной неэффективности. Во-вторых, делает бессмысленным использование тегов в указателях на постоянные объекты. И, наконец, по существу, возвращает понятие файла.

Временные указатели отнимать не надо, поэтому они в AS/400 существуют до ближайшей перезагрузки системы, что, так же является не самым лучшим решением.

Никаким другим целям, кроме указанных выше, теги в этой системе не служат. И уж во всяком случае, они совсем не обеспечивают межпроцедурную защищённость в смысле данной работы.

Ниже приведён более детальный технический анализ.

Ссылки (указатели)

1. Тегированная ссылка в AS/400 не содержит количества. Она может адресовать любой байт в сегменте, но пользовательская программа может свободно передвигать указатель в пределах сегмента. Таким образом, с точки зрения защиты указатель смотрит на весь сегмент.

Нельзя, например, передать в процедуру ссылку на одно слово или массив, находящиеся в сегменте, и занимающие только его часть, не передав этой процедуре право доступа до всего сегмента.

2. Как уже обсуждалось, проблема обеспечения владельцу сегмента возможности отъёма прав доступа других пользователей привела к необходимости обращаться при каждом доступе к справочнику пользователя, что крайне неэффективно.

Отъём нужен только в файлах (в связи с неограниченным временем жизни), а смешанное размещение ссылок и простых данных только в памяти (в связи с требованиями эффективности).

Отъём прав не представляет труда в разделённой системе, что продемонстрировано в описанной выше системе файлов Эльбрус.

В памяти, где отъём реализовывать не нужно, ссылки перемешаны.

В файлах же все ссылки сосредоточены в справочниках и в СВС файлов, как это сделано в Эльбрусе, поэтому отъём не представляет ни какого труда.

Работа с контекстом

Вызывает возражение обеспечение контекста работающей программы.
Каждый пользователь может создать набор справочников содержащих ссылки на доступные ему сегменты.

Запуская задачу (процесс) пользователь связывает с ним один из таких справочников. Всем программам, работающим в этом процессе (задаче), доступны все сегменты из данного справочника. Это неверно.

Как было показано выше, соответствие языковым принципам требует создание нового контекста при каждом запуске процедуры. Этот контекст состоит их данных приданных

процедуре в месте её объявления (условно, приданный автором процедуры), данных, переданных ей как параметры (переданных тем, кто её вызвал) и данных, созданных самой процедурой (локальные данные). И всё это должно работать эффективно.

Контекст, поддерживаемый системой справочников (техника системы файлов) с «авторизацией» указателей весьма громоздка. Создавать такой контекст при каждом запуске процедуры весьма накладно. Поэтому принято такое совсем не совершенное решение.

Правда для особо важных случаев есть специальные решения: допускающие доступ в вызванной программе к контексту, отличному от контекста процесса и наоборот, закрывающие доступ вызванной процедуре к контексту процесса.

Можно так сказать, что эти действия «имитируют» правильные решения, но они не решают правильно проблемы. Максимум, что стремились достичь авторы, это в условиях памяти единого уровня достичь защиты между различными задачами аналогичной той, которая существует в обычных системах.

Как и многое в системе защиты, переключение контекстов должно всегда выполняться в варианте максимальной защищённости и, при этом не требовать от программы (от программиста) каких либо специальных усилий. Программист должен заботиться о том, чтобы правильно решалась задача, а защита должна быть обеспечена автоматически.

Это может бать выполнено только в том случае, когда система полностью соответствует языковым основам.

Общий вывод

В системе AS/400, в отличие от Эльбруса, реализовано нечто среднее между традиционной системой защиты и защиты опирающейся на правильную работу с типами данных. В этом её основная слабость.

Её опыт показывает, что простого факта использования тегов совсем не достаточно, для создания совершенной системы защищённости. Для этого ещё необходимо, большое число правильных решений.

Тем не менее, эта система представляет, безусловно, положительное явление в вычислительном мире.

Микропроцессор iAPX 432

Эта система, выпущенная в 1981 году, сочетала в себе ортодоксально последовательное следование СпС подходу с рядом неправильных архитектурных решений.

Несмотря на предостережение разработчиков B5500 и опыт AS/400, говоривший о необходимости иметь смешанные сегменты, в iAPX 432 сегменты разделены по принципу *сегменты скаляров* и *сегменты ссылок*.

Даже в B5500, использовавшей СпС подход, в стек можно было загружать и ссылки и скаляры. В этой машине был и СпС подход, но были и теги! Этого разработчики iAPX 432 не заметили.

В результате эффективность в этой машине сильно пострадала.

Для запуска процедуры, например, у системы необходимо было просить два сегмента: один для ссылок, один для скаляров. В стеке выражений нельзя было держать смесь. В результате придумывались какие то невероятные ухищрения и оптимизации, сводившие на нет всю эффективность.

Самое печальное с этим проектом заключается в следующем,

Как показывает анализ, все коммерческие машины, рассмотренные здесь и выпущенные до iAPX 432, имели безусловный коммерческий успех, даже несмотря на значительные технические погрешности у многих из них.

Этот список включает B5000 и B5500, Plessey System 250, Эльбрус 1, 2, IBM S/38, AS/400.

Этот список можно пополнить и университетскими разработками, успешными в той степени, в какой могут быть успешными любые университетские проекты. Это машина Университета Rice, машина Айлифа и машина CAP (Кембридж).

Всё говорило о том, что это направление успешно развивается.

Однако после неудачи iAPX 432, все работы в этой области прекратились.

Вычислительный мир принял неуклюжесть конкретной разработки за неправильность направления.

Заключительные замечания по истории

Как уже отмечалось, вся история защищённой архитектуры берёт своё начало от работ Бартона (фирма Барроус) и Айлифа (Лондонский университет) в конце 50х и начале 60х.

Следующим событием, заметно повлиявшим на дальнейшее развитие этого направления, стала публикация Дениса и Ван Горна [3].

Как первые работы, так и эта публикация ориентировали дальнейшие работы на использование СпС подхода. Опыт первых машин (B5500) опубликованный авторами, разработки Эльбруса и IBM AS/400 убедительно показывают сомнительность такой ориентации, по меньшей мере, для реализации системы в памяти.

После первых реализаций многие коллективы вели успешные работы в этой области, было выпущено достаточное количество коммерческих и исследовательских машин и ОС.

Большинство реализаций опиралось на СпС подход. Единственным исключением является наиболее успешная за рубежом система AS/400, которая производится до настоящего времени. Эта система, как и Эльбрус, использует теги. Несмотря на ориентацию на теги, многие решения в этой системе не являются бесспорными. В частности, не лучшую роль сыграло использование памяти единого уровня.

Конец всем работам был положен крайне неудачной машиной iAPX 432, завершившейся неудачей.

К большому сожалению после этого события все фирмы (за исключением IBM и команды Эльбрус) и исследовательские организации прекратили работы в этой области.

Несколько слов о том, чем проект Эльбрус отличается от других работ.

В данной работе достаточно подробно приведены основания и детали архитектуры. Если попытаться представить всё это как можно короче, то можно сказать следующее.

В отличие от всех предыдущих работ, представленных в данном историческом обзоре, проект Эльбрус основан не на СпС подходе и, можно сказать, не на понятии **capability.**

Представляется, что, для правильной ориентации проекта, нет необходимости вводить какие то новые концепции. Всё очень хорошо укладывается в достаточно хорошо проработанные и широко известные понятия, такие как типы данных и строгий контроль правильности работы с ними. Всё то, что давно является базовой основой языков высокого уровня (к сожалению, не всегда правильно поддержанной реализацией).

Достаточно лишь точно и эффективно поддержать их реализацию в аппаратуре и программном обеспечении.

В основе должна быть реализована обычная языковая ссылка (указатель), свободно и эффективно создаваемая системой по просьбе любой программы вместе с объектом, ею адресуемым, имеющая свободу перемещения равную любым типам данных.

Ссылка должно уметь указывать на произвольные данные или их части, с произвольными правами доступа.

Для каждой процедуры, при её запуске, с помощью ссылок, формируется контекст, ограничивающий область доступных ей данных.

Всё это должно работать в соответствии с хорошо известной семантикой этих языковых понятий. Не меньше (без ограничений) и не больше (без нарушения защиты).

Вот и всё! Никаких **capability,** СпС подходов и т.д.

Только при этом условии программист ничего не будет знать про защищённость, но она будет самой совершенной.

Именно это и реализовано в Эльбрусе и именно это отличает его от всех систем описанных в этом историческом разделе.

Благодарности

В течение многих лет разработки основных принципов и нескольких поколений машин их реализующих достаточно много инженеров и программистов участвовали в этих работах. Сейчас трудно перечислить всех. Заранее следует извиниться перед теми, которые по вине автора не будут упомянуты здесь, что нисколько не означает оценки их вклада в изложенные работы.

В первоначальной оценке важности защищённости и разработке основных концепций по их реализации совместно с автором принимал участие Юлий Сахин, руководивший впоследствии всеми работами по их аппаратному воплощению.

Работами по программной реализации работ ведущая роль принадлежит Владимиру Волконскому и Сергею Семенихину. Первый в большей степени был связан с системами программирования и архитектурными вопросами, в особенности на этапе Эльбруса 3 – Е2 К, где он руководил всеми работами по программированию, второй работал в части обеспечения защищённости в операционной системе. Он руководил на Этапе Эльбруса 1, 2 созданием операционной системы Эльбрус, поддерживающей описываемый защищённый режим.

Реализацией исполняющих устройств (целых и вещественных) поддерживающих защищённость на всех этапах руководил Валерий Горштейн.

На первом этапе Эльбруса 1, 2 значительную роль в реализации систем программирования и некоторых архитектурных решений играл Владимир Пентковский, который, к сожалению, совсем не участвовал в дальнейших работах (этап Эльбруса 3 – Е2К). Он руководил разработкой базового языка Эль 76 для машин Эльбрус 1, 2.

Важную роль на всех этапах в глубоком осмыслении и аппаратной реализации всех этих идей принадлежит Фёдору Груздову.

Определяющую роль на втором этапе в программной реализации этих работ играли Валентин Тихонов и Евгений Эльцин.

Большую благодарность автор выражает Денису Хартикову, который взял на себя труд по прочтению рукописи, и чьи ценные замечания помогли её улучшению.

И, наконец, огромную благодарность автор выражает Любови Гладких и её сотрудникам, затративших много усилий по работе над текстом статьи и его переводу.

Литература

[1] Henry M. Levy. *Capability-Based Computer Systems.* Digital Press, Bedford, Mass., USA, 1984.

[2] Frank G. Soltis. *Inside the AS/400 (second edition).* Duke Press, Loveland, Colorado, USA, 1997.

[3] J. B. Dennis and E. C. Van Horn. Programming Semantics for Multiprogrammed Computations. *Communications of the ACM 9(3),* March 1966.

[4] *The Descriptor – a Definition of the B5000 Information Processing System.* Burroughs Corporation, Detroit, Michigan, 1961.

[5] M. E. Houdek, F. G. Soltis, and R. L. Hoffman. IBM System/38 Support for Capability-Based Addressing. In *Proceedings of the 8th Symposium on Computer Architecture*. ACM/IEEE, May 1981.

[6] F. G. Soltis and R. L. Hoffman. Design Considerations for the IBM System/38. In *Proceedings of Compcon 79. Spring 1979.*

[7] W. A. Wulf, R. Levin, and S. P. Harbison. *HYDRA/C.mmp: An Experimental Computer System.* McGraw-Hill, New York, 1981.

[8] M. V. Wilkes and R. M. Needham. *The Cambridge CAP Computer and its Operating System.* New York, 1979.

[9] D. M. England. Architectural Features of System 250. In *Infotech State of the Art Report on Operating Systems.* Infotech, 1972.

[10] E. A. Feustel. The Rice research Computer – A Tagged Architecture. In *Proceedings of the Spring Joint Computer Conference,* pages 369-377. IFIP, 1972.

[11] A. K. Jones, R. J. Chansler, Jr. I. Durham, K. Schwans and S. R. Vegdahl. StarOS, A Multiprocessor Operating System for the Support of Task Forces. In *Proceeding of the 7th Symposium on Operating Systems Principles,* pages 117-127. December 1978.

[12] iAPX 432 *General Data Processor Architecture Reference Manual.* Revision 3 (Advance Partial Issue) edition, Santa Clara, California, 1982.

[13] J. K. Iliffe. *Basic Machine Principles.* American Elsevier, Inc., New York, 1968.

[14] V. H. Yngve. The Chicago Magic Number Computer. In *ICR Quarterly Report,* pages B1-B20. U. of Chicago Institute for Computer Research, November 1968.

[15] www.elbrus.ru

[16] www.kaspersky.com