РЕФЕРАТ

Расчетно-пояснительная записка к выпускной квалификационной работе на тему «Реализация очистки программного и аппаратного стека для архитектуры микропроцессора „Эльбрус”» содержит 88 страниц машинописного текста, 9 рисунков, 3 таблицы.

Цель работы – реализация процесса сборки мусора для архитектуры микропроцессора «Эльбрус».

Поставленная цель достигается путем реализации доступа ко всем сегментам памяти, используемых во время работы процесса. Для решения поставленной задачи учитывались все особенности архитектуры микропроцесса и операционной системы.

Для проверки работоспособности функционала портируемого сборщика мусора проведены тестирования и анализ полученных результатов путем сравнения с реализациями на других архитектурах. По завершении адаптации процесса сборки под новую систему реализована кроссплатформенная сборка программного пакета для архитектур микропроцессоров SPARC, x86 и E2K. Проведена эксплуатация сборщика мусора на стороннем приложении. На основе результатов профилирования работы приложения выявлены временные задержки при выполнении сборки мусора. Для ускорения работы сборщика мусора осуществлена оптимизация решения с учетом особенностей архитектуры микропроцессора. Повторное профилирование работы приложения, использующего реализованный процесс сборки мусора с новыми модификациями, показало уменьшение временных затрат, приходившихся на очистку памяти.

СОДЕРЖАНИЕ

РЕФЕРАТ 6

ВВЕДЕНИЕ 9

Определения, обозначения и сокращения 11

1 Анализ предметной области 13

1.1 Процесс сборки мусора 13

1.2 Требования к языку и системе 18

1.3 Достоинства и недостатки автоматической сборки мусора 19

1.4 Реализация алгоритма для языков программирования C/C++ 22

1.5 Реализация доступа к памяти для сборки мусора 27

1.6 Особенности микропроцессорной архитектуры SPARC 29

1.7 Особенности микропроцессорной архитектуры «Эльбрус» 35

2 Постановка задачи 41

2.1 Постановка задачи организации доступа к памяти для архитектуры микропроцессора «Эльбрус» 41

2.2 Тестирование реализации алгоритма и анализ результатов 41

2.3 Кроссплатформенная сборка 42

2.4 Оптимизация решения 42

3 Изучение реализованных методик решений 44

3.1 Описание существующих аналогов 45

3.2 Особенности организации памяти микропроцессора «Эльбрус» .. 55

4 Реализация сборки мусора для новых платформ 61

4.1 Принцип платформенных реализаций 61

4.2 Структура архитектурно-зависимых частей 62

4.3 Описание операционной системы и процессора в заголовочных файлах (gcconfig.h, gc_priv.h, gc.h) 64

4.4 Реализация доступа к процедурному стеку для архитектуры «Эльбрус» 67

4.5 Определение границ обрабатываемой памяти (os_dep.c)..... 71

4.6 Передача данных о границах процедурного стека (mach_dep.c)... 72

4.7 Операция «Остановки мира» (pthread_stop_world.c) 72

4.8 Маркировка участков памяти (mark_rts.c) 73

5 Тестирование реализации и анализ результатов 74

5.1 Проведение внутренних тестов сборщика мусора..... 74

5.2 Сравнение результатов на Эльбрусе и на Intel 75

5.3 Выводы по проведенным тестированиям 75

6 Кроссплатформенная сборка 76

6.1 Особенности кроссплатформенной сборки для архитектуры «Эльбрус» 76

6.2 Уровни оптимизации компилятора..... 77

6.3 Сборка под различные архитектуры (E2K, x86, SPARC)..... 79

7 Использование и дальнейшая оптимизация решения..... 81

7.1 Применение сборщика мусора на сторонних приложениях 81

7.2 Оптимизация решения на основе результатов эксплуатации 83

7.3 Результаты проведенной оптимизации 85

ЗАКЛЮЧЕНИЕ..... 86

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 87

ПРИЛОЖЕНИЕ А. Графическая часть магистерской диссертации..... 89

ВВЕДЕНИЕ

Память является одним из самых важных ресурсов компьютера, и в силу того, что она имеет ограничения по размеру, программисту необходимо предусматривать способы ее очистки.

Первичным вариантом для освобождения ресурсов является ручное управление памятью. Сущность метода заключается в том, что программист для создания объекта в динамической памяти явно вызывает команду выделения памяти. Данная команда возвращает указатель на выделенную область памяти, который сохраняется и используется для доступа к ней. До тех пор, пока объект нужен для работы программы, программа обращается к нему через ранее сохраненный указатель. Когда надобность в объекте проходит, то программист явно вызывает команду освобождения памяти, передавая указатель на удаляемый объект. В любом языке, допускающем создание объектов в динамической памяти, потенциально возможны две проблемы: висячие ссылки и утечки памяти.

Висячие ссылки – это оставшаяся в использовании ссылка на объект, который уже удален. После удаления объекта все сохранившиеся в программе ссылки на него становятся «висячими». Память, занимаемая ранее объектом, может быть передана операционной системе и стать недоступной, или быть использована для размещения нового объекта в той же программе. В первом случае попытка обратиться по «повисшей» ссылке приведет к срабатыванию механизма защиты памяти и аварийной остановке программы, а во втором – к непредсказуемым последствиям.

Создав объект в динамической памяти, программист может не удалить его после завершения использования. Если ссылающийся на объект переменной будет присвоено новое значение и на объект других ссылок нет, он становится программно недоступным, но продолжает занимать память, поскольку команда его удаления не вызывалась. Данная ситуация называется утечкой памяти.

Приведенные проблемы ручного управления памятью послужили основой для создания автоматического процесса, который бы решал данные задачи. Сборка мусора – технология, которая позволяет с одной стороны, упростить программирование, избавив программиста от необходимости вручную удалять объекты, созданные в динамической памяти, с другой – устранять ошибки, вызванные неправильным ручным управлением памятью.

Если бы память компьютера была бесконечной, можно было бы просто оставлять ненужные объекты в памяти. Процесс сборки мусора является эмуляцией такого бесконечного компьютера на конечной памяти. Из этого следует: если во время загрузки программы доступно больше оперативной памяти, чем может потребоваться для работы, то сборщик мусора не будет выполнять процесс по очистки памяти. Таким образом, менеджер памяти будет выделять память программе, как только она потребуется, и, по условиям, это выделение всегда будет успешным, то есть, если компьютер имеет больше оперативной памяти, чем это требуется программе, то эмуляция бесконечной памяти не нужна.

В системе со сборкой мусора обязанность освобождения памяти от объектов, которые больше не используются, возлагаются на среду исполнения программы. Программист лишь создает динамические объекты и пользуется ими. Все вопросы, связанные с удалением объектов, решаются средой разработки. Для осуществления сборки мусора в состав среды исполнения включается специальный программный модуль, называемый сборщиком мусора. Этот модуль периодически запускается, определяет, какие из созданных в динамической памяти объектов более не используется, и по необходимости освобождает занимаемую ими память.

В силу того, что сборщик мусора работает напрямую с памятью компьютера, такому модулю памяти необходимо иметь прямой доступ аппаратным и процедурным стекам памяти. Организация памяти в компьютере напрямую зависит от архитектуры процессора и операционной системы.

Определения, обозначения и сокращения

ALU – Arithmetic Logic Unit – арифметико-логическое устройство

APB – Array Prefetch Buffer – буфер предварительной подкачки массивов

API – Application Programming Interface – интерфейс программирования приложений

ASI – Address Space Identifier – идентификатор адресного пространства

CISC – Complex Instruction Set Computer – компьютер с полным набором команд

CPU – Central Processor Unit – центральный процессор

CS – Code Segment – указатель на кодовый сегмент

CU – Control Unit – устройство управления

DMA – Direct Memory Access – прямой доступ к памяти

DS – Data Segment – сегмент данных

GC – Garbage Collector – сборщик мусора

IB – Instruction Buffer – буфер команд

IP – Instruction Pointer – регистр, указывающий на смещение (адрес) инструкций в сегменте кода

POSIX – Portable Operating System Interface for UniX – интерфейс переносимых операционных систем UNIX

RAM – Random Access Memory – память с произвольным доступом

RF – Register File – регистровый файл

RISC – Reduced Instruction Set Computer – архитектура компьютера с сокращенным набором команд

SPARC – Scalable Processor Architecture – масштабируемая процессорная архитектура

SP – Stack Pointer – указатель стека

SS – Stack Segment – указатель на сегмент стека

TU – Trap Unit – устройство обработки прерываний

VLIW – Very Long Instruction Word **–** широкая команда

АВС – аппаратурная вершина стека

БК – буфер команд

ВС – вычислительная система

МВК – многопроцессорный вычислительный комплекс

МЦСТ – Московский центр SPARC-технологий

ОПО – общее программное обеспечение

ОС – операционная система

ОСПО – общесистемное программное обеспечение

ПС – программное средство

ПФ – предикатный файл

РгФ – регистровый файл

СПО – системное программное обеспечение

СОЗУ – сверхбыстрая оперативная память

УВС – указатель вершины стека

УЛП – устройство логических предикатов

УОМ — устройство обращения к массивам

УОП — устройство обращения в память

УУ – устройство управления

УУП — устройство управления памятью

ЦП – центральный процессор

ШК – широкая команда

1 Анализ предметной области

1.1 Процесс сборки мусора

Процесс сборки мусора является автоматическим решением очистки оперативной памяти компьютера для корректной работы программ. Данная технология предоставляет возможность программисту переложить работу с памятью на автоматизированный процесс среды разработки. Для осуществления сборки мусора в состав среды исполнения включается специальной программный модуль, называемый «сборщиком мусора». Этот модуль периодически запускается, определяет, какие из созданных в динамической памяти объектов более не используются, и освобождает занимаемую ими память.

Периодичность запуска сборщика мусора определяется особенностями системы. Сборщик может работать в фоновом режиме, запускаясь при неактивности программы (например, когда программа простаивает, ожидая ввода данных пользователем). Сборщик мусора запускается безусловно, прерывая на время своей работы выполнение программы, когда очередную операцию выделения памяти оказывается невозможно выполнить из-за того, что вся доступная память исчерпана. После освобождения памяти прерванная операция выделения памяти возобновляется, и программа продолжает исполняться дальше. Если же оказывается, что освободить память невозможно, среда исполнения останавливает программу с сообщением об ошибке: «Недостаточно памяти».

Хотя, в общем случае, невозможно точно определить момент, когда объект был использован в последний раз и больше не нужен, сборщики мусора используют консервативные оценки, позволяющие определить, что в будущем объект гарантированно не будет использоваться.

Обычно критерием того, что объект ещё используется, является наличие ссылок на него. Если в системе нет больше ссылок на данный объект, то он,

очевидно, больше не может быть использован программой, а, следовательно, может быть удалён. Этот критерий используется большинством современных сборщиков мусора и называется ещё достижимостью объекта.

Неформально можно задать следующее рекурсивное определение достижимого объекта:

1. Определённое множество объектов считается достижимым изначально — корневые объекты, обычно в их число включают все глобальные переменные и объекты, на которые есть ссылки в стеке вызовов;

2. Любой объект, на который есть ссылка из достижимого объекта, тоже считается достижимым, за исключением ссылок, указанных программистом как слабая.

Такое определение не является теоретически наилучшим, так как в число достижимых, согласно ему, попадают и те объекты, которые уже никогда не будут использованы, но на которые пока ещё существуют ссылки. Оптимальным было бы считать недостижимым объект, к которому в процессе дальнейшей работы программы не будет ни одного обращения, однако выявление таких объектов невозможно, поскольку сводится к алгоритмически неразрешимой задаче об остановке (для этого достаточно предположить, что некоторый объект X будет использован в том и только в том случае, если успешно завершится программа P).

Для определения достижимости объекта существует алгоритм выставления флагов. Данное решение встречается также с названием «Алгоритм пометок» (Mark and Sweep), заключается в следующем:

- для каждого объекта хранится бит, указывающий, достижим ли этот объект из программы или нет;

- изначально все объекты, кроме корневых, помечаются как недостижимые;

- рекурсивно просматриваются и помечаются как достижимые объекты, ещё не помеченные, и до которых можно добраться из корневых объектов по ссылкам;

- те объекты, у которых бит достижимости не был установлен, считаются недостижимыми.

Следует обратить внимание, что, согласно данному алгоритму, если два или более объектов ссылаются друг на друга, но ни на один из этих объектов нет других ссылок, то имеет место циклическая ссылка, и вся группа считается недостижимой. Эта особенность алгоритма позволяет гарантированно удалять группы объектов, использование которых прекратилось, но в которых имеются ссылки друг на друга. Такие группы часто называются «islands of isolation» (острова изоляции).

Другой вариант алгоритма определения достижимости — обычный подсчёт ссылок. Его использование замедляет операции присваивания ссылок, но зато определение достижимых объектов тривиально — это все объекты, значение счётчика ссылок которых превышает нуль. Без дополнительных уточнений этот алгоритм, в отличие от предыдущего, не удаляет циклически замкнутые цепочки вышедших из употребления объектов, сохранивших ссылки друг на друга.

Как только определено множество недостижимых объектов, сборщик мусора может освободить память, занимаемую ими, и оставить остальное как есть. Также можно после освобождения памяти переместить все или часть оставшихся объектов в другие области памяти, обновив вместе с этим все ссылки на них. Эти два варианта реализации называются, соответственно, *неперемещающим* и *перемещающим*.

Обе стратегии имеют как достоинства, так и недостатки:

1. *Скорость выделения и освобождения памяти.* Неперемещающий сборщик мусора быстрее освобождает память (поскольку эта операция сводится к пометке соответствующих блоков памяти как свободных), но тратит больше времени на её выделение (поскольку память фрагментируется, и при выделении необходимо найти в памяти нужное количество блоков подходящего размера).

Перемещающий сборщик требует сравнительно больше времени при сборе мусора (тратится дополнительное время на дефрагментацию памяти и изменение всех ссылок на перемещаемые объекты), зато перемещение позволяет использовать чрезвычайно простой и быстрый (O(1)) алгоритм выделения памяти. При дефрагментации объекты передвигаются так, чтобы разделить всю память на две большие области — занятую и свободную, и сохраняется указатель на их границу. Для выделения новой памяти достаточно лишь переместить эту границу, вернув кусок из начала свободной памяти.

2. *Скорость доступа к объектам в динамической памяти.* Объекты, поля которых используются совместно, перемещающий сборщик позволяет размещать в памяти недалеко друг от друга. Тогда они вероятнее окажутся в кэше процессора одновременно, что уменьшит количество обращений к относительно медленному ОЗУ.

3. *Совместимость с инородным кодом.* Перемещающий сборщик мусора вызывает затруднения при использовании кода, который не контролируется системой автоматического управления памятью, такой код называется инородным (англ. foreign) в традиционной терминологии или неуправляемым (англ. unmanaged) в терминологии Microsoft. Указатель на память, выделенную в системе с неперемещающим сборщиком, можно просто передать инородному коду для использования, удерживая хотя бы одну обычную ссылку на объект, чтобы сборщик его не удалил. Перемещающий сборщик меняет положение объектов в памяти, синхронно меняя все ссылки на них, но поменять ссылки в инородном коде он не может, в результате переданные инородному коду ссылки после перемещения объекта станут некорректными. Для работы с инородным кодом используются различные специальные приёмы, например, прикрепление (англ. pinning) — явная блокировка объекта, запрещающая его перемещение во время сборки мусора.

Существуют механизмы сборки мусора, в которых предусмотрен учет поколений объектов. Как показывает практика, недавно созданные объекты чаще становятся недостижимыми, чем объекты, существующие длительное время. В соответствии с этой закономерностью многие современные сборщики мусора подразделяют все объекты на несколько поколений — серий объектов с близким временем существования. Как только память, выделенная одному из поколений, заканчивается, в этом поколении и во всех более «молодых» производится поиск недостижимых объектов. Все они удаляются, а оставшиеся переводятся в более «старое» поколение.

Использование поколений сокращает время цикла сборки мусора, поскольку уменьшается число просматриваемых в ходе сборки объектов, однако этот метод требует от среды исполнения отслеживания ссылок между разными поколениями.

В зависимости от языка программирования предусматриваются также другие механизмы сборки мусора, учитывающие непосредственные особенности языка разработки. Например, в Java существуют неизменяемые объекты (англ. immutable objects): java.lang.String – как только строке присвоили какое-то значение, то его уже нельзя изменить. Подпрограммы будут передавать ссылку на эту строку друг другу, и когда все ссылки исчезнут, строка будет уничтожена сборщиком мусора.

Некоторые алгоритмы сборщика мусора предусматривают *финализаторы*. Финализатор указывает, что делать, когда объект попадает под сборщик мусора. Обычно финализаторы пишут для обёрток над объектами операционной системы (файлами, сетевыми сокетами); они автоматически закрывают соответствующий объект.

Поскольку объект до сбора может «провисеть» в памяти довольно долго, хорошая манера программирования — закрыть файл или сокет вручную, командой наподобие close().

1.2 Требования к языку и системе

Для того, чтобы программа могла использовать сборку мусора, необходимо выполнение ряда условий, относящихся к языку, среде исполнения и самой решаемой задаче:

1. *Необходимость среды исполнения со сборщиком мусора.* Естественно, для сборки мусора необходима динамическая среда, поддерживающая исполнение программы, и наличие в этой среде сборщика мусора.

2. *Поддержка со стороны языка программирования.* Сборщик мусора может нормально функционировать только тогда, когда он может точно отследить все ссылки на все созданные объекты. Очевидно, если язык допускает преобразование ссылок (указателей) в другие типы данных (целые числа, массивы байтов и так далее), такой как Си/Си++, отследить использование таких преобразованных ссылок становится невозможно, и сборка мусора становится бессмысленной — она не защищает от «повисания» ссылок и утечек памяти. Поэтому языки, ориентированные на использование сборки мусора, обычно существенно ограничивают свободу использования указателей, адресной арифметики, преобразований типов указателей к другим типам данных. В части из них вообще нет типа данных «указатель», в части он есть, но не допускает ни преобразований типа, ни изменения[1].

3. *Техническая допустимость кратковременных замедлений в работе программ.* Сборка мусора выполняется периодически, как правило, в заранее неизвестные моменты времени. Если приостановка работы программы на время, сравнимое со временем сборки мусора, может привести к критическим ошибкам, использовать в подобной ситуации сборку мусора, очевидно, нельзя.

4. *Наличие некоторого резерва свободной памяти.* Чем больше памяти доступно среде исполнения, тем реже запускается сборщик мусора и тем эффективнее его работа. Работа сборщика мусора в системе, где количество доступной сборщику памяти приближается к пиковой потребности программы, может оказаться неэффективной и непроизводительной. Чем меньше избыток

памяти, тем чаще происходит запуск сборщика и тем больше времени тратится на его выполнение. Падение производительности программы в таком режиме может оказаться слишком существенным.

Сборка мусора часто противопоставляется ручному управлению памятью, при котором программист явно указывает, когда и какие области памяти надо освободить. Однако есть языки, в которых используется комбинация двух методов управления памятью, равно как есть и другие технологии решения той же фундаментальной проблемы (например, en:region inference).

Некоторые языки программирования требуют использования механизма сборки мусора в соответствии со своей спецификацией (Java, C#, Eiffel), другие — по причинам эффективности реализации (например, формальные языки для лямбда-исчисления) — эти языки называются языками со сборкой мусора. Многие языки со сборкой мусора не имеют возможностей для явного ручного удаления объектов (например, оператора delete), благодаря чему возникновение висячих ссылок исключается в принципе, а сборщик мусора лишь занимается удалением объектов, на которые нет ссылок из программы.

Некоторые языки, например, Modula-3, позволяют использовать как ручное управление памятью, так и сборку мусора в одном приложении — используя две отдельные кучи.

1.3 Достоинства и недостатки автоматической сборки мусора

По сравнению с ручным управлением памятью сборка мусора безопаснее, поскольку она предотвращает утечки памяти и возникновение висячих ссылок из-за несвоевременного удаления объектов. Другой положительный момент — упрощение самого процесса программирования. С другой стороны, наличие сборки мусора может вызывать у неопытного разработчика чувство ложной безопасности, базирующееся на представлении, что вопросам выделения и освобождения памяти вообще не надо уделять внимания, поскольку они решаются сборщиком мусора.

Например, объект никогда не будет удалён, если на него остался хотя бы один необнулённый указатель в глобальной области видимости, и поиск такой псевдоутечки в языках со сборщиком мусора особенно сложен. Программист не может полностью игнорировать вопрос управления памятью при наличии сборщика мусора, хотя затраты ручного труда на управление памятью в этом случае всё-таки существенно меньше по сравнению с языками с полностью ручным управлением (без сборщика и автодеструкторов). Зачастую критически важной является не только гарантия освобождения ресурса, но и гарантия того, что он освободится до вызова какой-то другой процедуры — например, открытые файлы, входы в критические секции. Отдавать управление этими ресурсами сборщику мусора нельзя, поэтому приходится убирать их вручную. Впрочем, в последнее время даже в языках со сборщиком мусора вводят возможность создавать классы с детерминированным вызовом специального метода-«деструктора» (Dispose) при выходе из зоны видимости [2].

Во многих случаях системы со сборкой мусора демонстрируют меньшую эффективность, как по скорости, так и по объёму используемой памяти (что неизбежно, так как сборщик мусора сам потребляет ресурсы и нуждается в некотором избытке свободной памяти для нормальной работы). Кроме того, в системах со сборкой мусора сложнее реализуются низкоуровневые алгоритмы, требующие прямого доступа к оперативной памяти компьютера, поскольку свободное использование указателей невозможно, и прямой доступ к памяти требует наличия специальных интерфейсов, написанных на низкоуровневых языках. С другой стороны, в современных системах со сборкой мусора операция выделения памяти сведена к элементарному добавлению блока в конец кучи, причём куча время от времени уплотняется, уменьшая фрагментацию данных.

Поддержка в некоторых императивных языках автоматического вызова деструктора (C++, Ада, Delphi), а также более простая, чем сборка мусора, технология использования «умных ссылок» (отслеживания количества ссылок

на объект непосредственно в нём и автоматическое удаление при удалении последней ссылки, как это сделано в технологии COM) значительно снижает вероятность утечек, позволяя концентрировать опасные места внутри реализации класса, при этом не требуя лишних ресурсов, хотя и требует при этом более высокой квалификации программиста. Конечно, писать код освобождения ресурсов всё равно придётся, но автоматические деструкторы дадут уверенность, что этот код обязательно вызовется. Впрочем, для наиболее часто используемых ресурсов во всех популярных языках, поддерживающих автодеструкторы, уже есть автоматические обёртки, которые сами закрывают ресурс, благодаря чему забота о ресурсах становится едва ли не проще, чем с непредсказуемым сборщиком мусора.

Существенное удобство от сборки мусора возникает тогда, когда динамически созданные объекты живут длительное время, многократно дублируются, а ссылки на них передаются между различными участками программы. В таких программах в общем случае достаточно сложно безошибочно определить место, где объект перестал использоваться и его можно удалять. Поскольку именно такая ситуация складывается при широком использовании динамически изменяемых структур данных (списки, деревья, графы), сборка мусора является необходимой в широко использующих такие структуры функциональных и логических языках, типа Хаскелла, Лиспа или Пролога. Использование сборки мусора в традиционных императивных языках (основанных на структурной парадигме, возможно, дополненной объектными средствами) определяется желаемым соотношением между простотой и скоростью разработки программ и эффективностью её выполнения.

Вопреки часто встречающимся утверждениям, наличие сборки мусора вовсе не освобождает программиста от всех проблем управления памятью.

Например, возникают трудности с освобождением других ресурсов, занятых объектом. Помимо динамической памяти, объект может владеть и другими ресурсами — подчас более ценными, чем память. Если объект при создании открывает файл, по завершении использования он должен его

закрыть, если подключается к СУБД — должен отключиться. В системах с ручным управлением памятью это делается непосредственно перед удалением объекта из памяти, чаще всего — в деструкторах соответствующих объектов. В системах со сборкой мусора обычно есть возможность выполнить некоторый код непосредственно перед удалением объекта, так называемые финализаторы, но для освобождения ресурсов они не годятся, так как момент удаления заранее неизвестен, и может оказаться, что ресурс освобождается намного позже, чем перестаёт использоваться объект. В подобных случаях программисту всё равно приходится отслеживать использование объекта вручную и вручную выполнять операции по освобождению занятых объектом ресурсов. В C# специально для этой цели существует интерфейс IDisposable, в Java — Closeable.

В системах со сборкой мусора тоже могут возникать утечки памяти, правда, имеющие несколько другую природу. Ссылка на неиспользуемый объект может сохраниться в другом объекте, который используется и становится своеобразным «якорем», удерживающим ненужный объект в памяти. Например, созданный объект добавляется в коллекцию, используемую для вспомогательных операций, потом перестаёт использоваться, но не удаляется из коллекции. Коллекция удерживает ссылку, объект остаётся достижимым и не подвергается сборке мусора. Результатом становится всё та же утечка памяти [3], [4].

Чтобы устранить подобные проблемы, среда исполнения может поддерживать специальное средство — так называемые слабые ссылки. Слабые ссылки не удерживают объекта и превращаются в null, как только объект исчезает — поэтому код должен быть готов к тому, что однажды ссылка укажет в никуда.

1.4 Реализация алгоритма для языков программирования C/C++

Алгоритм сборки мусора для языков программирования C/C++ был впервые реализован американским программистом Ханцем Боэмом (англ. Hans-

Juergen Boehm) совместно с профессором компьютерных технологий Аланом Демерсом (англ. Alan Demers). Данная реализация сборщика мусора предоставляет программисту возможность переложить ответственность за освобождение памяти на среду использования. Таким образом, сборщик мусора решает, как одну из частных задач об устранении утечек памяти.

В основе реализации сборки мусора лежит модифицированный алгоритм пометок (англ. modified mark-sweep algorithm) для определения достижимости объектов. Непосредственная работа сборщика мусора начинается в момент остановки работы всех потоков. Данная реализация процесса описывается принципом «остановки мира» (англ. pthreads stop the world).

Работа сборщика мусора определяется четырьмя основными фазами алгоритма:

1. *Подготовка.* Каждый объект имеет соответствующую маркировку бита. Производится удаление всех маркировок битов, а затем инициализация всех созданных объектов как потенциально недостижимые.

2. *Фаза маркировки.* Осуществляется маркировка всех объектов, которые могут быть достижимы с помощью цепочки указателей от переменных. Часто сборщик мусора не имеет никакой реальной информации о местонахождении указателей переменных в массиве данных, поэтому производится рассмотрение всех статических областей данных, стеков и регистров как потенциально содержащих указатели. Любые комбинации битов, которые представляют адреса внутри массива данных объектов, рассматриваются сборщиком мусора как указатели. Если существуют объекты недоступные во время фазы маркировки, то на момент получения доступа переменные сканируются аналогичным образом [5].

3. *Фаза очистки.* Производится сканирование массива данных на предмет недоступных, то есть не промаркированных, объектов и возвращает их в соответствующий свободный список для повторного использования. На самом деле это не отдельная фаза, даже в не пошаговом режиме эта операция обычно выполняется по требованию, когда обнаруживается, что свободный

список пуст. Таким образом, крайне маловероятно, что на фазе очистки будут рассмотрены объекты, которые были не рассмотрены ранее.

4. *Финальная фаза.* Недоступные объекты, которые были зарегистрированы для окончательной доработки становятся в очередь для финального удаления сборщиком.

Сборщик включает в себе свой собственный распределитель памяти. Распределитель получает память в системе в зависимости от платформы. В UNIX используется malloc, sbrk или mmap.

Распределителем используются большинство статических данных, а все необходимое для остальной части сборщика мусора хранится внутри _GC_array структуры. Такая организация данных позволяет сборщику мусора легко игнорировать собственные структуры данных при поиске корневых указателей. Другая часть распределителя сборщика внутренней структуры данных динамически выделяются в GC_scratch_alloc. GC_scratch_alloc не допускает освобождение памяти и поэтому используется для постоянных структур.

Распределитель распределяет объекты разных видов. Разные виды обрабатываются несколько иначе определенными частями сборщика мусора.

Определенные виды проверяются на наличие указателей, другие нет. Некоторые из них могут иметь дескрипторы типа отдельных объектов, которые определяют местоположение указателя, или конкретный вид может соответствовать одной конкретной компоновке объектов. Есть два встроенных вида, которые являются невозвратными. Один из которых (STUBBORN) неизменяемый без особых мер предосторожности. Несмотря на то, что в основном используются два вида объектов: NORMAL и PTRFREE.

Сборщик использует два уровня распределителя. Один большой блок определяется размером большим чем половина HBLKSIZE. Большие блоки округляются до следующего кратного HBLKSIZE, а затем выделяются в GC_allochblk. Малые блоки выделяются размером HBLKSIZE. Каждый блок рассчитан только на размер одного объекта и вида.

После того, как большой блок разделен для использования небольших объектов, он может быть использован только для объектов этого размера, если сборщик не обнаруживает совершенно пустой блок. Полностью пустые части восстанавливаются в соответствующий большой блок свободного списка.

Для того, чтобы избежать большого выделения блоков под объекты различных размеров, сборщик выделяет блоки под объекты исходя из возможных размеров. Вместо запроса на новый блок размер объекта округляется до возможного из выделенных размеров, для которых поддерживаются свободные списки. Точные выделенные размеры определяются по запросу, но при условии невозможного выполнения, размер блока увеличивается в геометрической прогрессии. Таким образом, объекты, запрошенные в начале исполнения, вероятно, будут выделены с точным требуемым размером, с учетом ограничения выравнивания.

На этапе маркировки сборщик помечает все объекты, которые возможно достижимы из переменных указателей. Так как невозможно определить где переменные указатели расположены, сборщик сканирует следующие сегменты для корневых указателей:

1. Регистры. В зависимости от архитектуры возможна реализация с помощью кода сборки или путем вызова setjmp как функцию, которая сохраняет содержимое регистра в стеке [8].

2. Стеки. В случае однопоточного приложения, на большинстве платформ реализуется путем сканирования памяти и между приближенного значения текущего указателя стека и GC_stackbottom. Переменная GC_stackbottom устанавливается в зависимости от платформы системы и описывается непосредственно в файле конфигурации gcconfig.h.

3. Статические области данных. В простейшем случае эта область между DATASTART и DATAEND, как это определено в gcconfig.h. Тем не менее в большинстве случаев, в себе будет также включать статические области данных, связанных с динамическими библиотеками, которые определяются для большинства платформ в dyn_load.c.

Маркировка поддерживает явный стек области памяти, который должен быть доступный для поиска указателей. Каждая запись в стеке содержит начальный адрес блока для сканирования, а также дескриптор блока. Если информация из блока недоступна, то дескриптор содержит длину [9, 11].

В начале фазы маркировки все корневые сегменты отправляются в стек GC_push_roots. Если ALL_INTERIOR_PTRS не определен, то корневой стек требует специальной обработки. В этом случае нормальная маркировка кода игнорирует внутренние указатели, но GC_push_all_stack явно проверяет наличие внутренних указателей и получает дескрипторы для целевых объектов.

Маркировка построена так, что позволяет осуществить инкрементальную маркировку. Каждый вызов GC_mark_some выполняет небольшой объем работы для маркировки массива данных. Состояние маркера отображается в поле GC_mark_state, которое идентифицирует конкретную подфазу. Другие поля определяют, сколько работы еще предстоит сделать в каждой подфазе. Стандартная прогрессия маркировки в фазе «остановки мира» протекает по следующим этапам:

1. MS_INVALID указывает на доступ к немаркированным объектам. В этом случае GC_objects_are_marked будет содержать ложное значение, так что указатель передвигается к MS_PUSH_UNCOLLECTABLE.

2. MS_PUSH_UNCOLLECTABLE указывает, что достаточно определить корневые объекты, а затем отметить все достижимые из них. Scan_ptr продвигается через массив данных пока не определятся все недостижимые объекты, а остальные достижимые не будут отмечены.

3. MS_ROOTS_PUSHED указывает, что как только массив данных окажется пуст, то все доступные объекты помечены. По завершению состояние меняется на MS_NONE.

4. MS_NONE указывает, что достижимые объекты отмечены.

Фаза «остановки мира» подразумевает, что работа программы приостановлена и, соответственно, и работа потоков тоже.

По окончанию фазы маркировки рассматриваются все блоки в массиве данных. Немаркированные объекты помещаются в большой блок свободного списка. Каждый малый объект проверяется, если все определено верно, то объект возвращается в большой блок, иначе помещается в блок для повторной проверки достижимости. Следующая фаза рассматривается только в том случае, если необходимо освобождение памяти.

Объекты разделяются на два типа: неиспользуемые, которые будут в дальнейшем удалены, и активные, которые следует переместить для освобождения памяти. Для осуществления перемещения необходимо копирование блоков памяти и обновление ссылок на них. Данные операции являются достаточно затратными по выполнения.

Все неиспользуемые объекты удаляются по необходимости освобождения памяти.

1.5 Реализация доступа к памяти для сборки мусора

Для осуществления работы сборщика мусора необходимо передавать как входные данные информацию об особенностях организации памяти данной архитектуры процессора, а также непосредственно явные адреса участков, для которых необходимо совершить процесс сборки мусора.

В силу того, что очистка памяти зависит от типа процессора, его особенностей, а также от используемой операционной системы, то в сборщике мусора, реализованном Хансом Боэмом, предусмотрены архитектурно-зависимые файлы, по средствам которых определяются все особенности среды использования [10].

В качестве решения для осуществления кроссплатформенной сборки необходимые характеристики используемой архитектуры указываются в специальном заголовочном файле. Данное решение предусматривает достаточно простое портирование сборщика мусора под другие архитектуры.

Конфигурационной файл состоит из трех основных частей, в котором при помощи #ifdef директив определяются главные характеристики процессора и операционной системы:

1. Раздел, определяющий основные внутренние макросы сборщика мусора, а именно архитектуру процессора (например, ia64, i386) и операционную систему (например, LINUX). Данное переопределение макросов позволяет изолировать сборку от различий компилятора.

2. Раздел, который определяет наибольшее количество платформенных макросов, которые в дальнейшем будут использоваться компилятором.

3. Раздел, который заполняет по умолчанию неопределенные ранее платформенные макросы и определяет детальные особенности сборки.

Заголовочный файл, в котором описываются особенности архитектуры системы, также содержит информацию о возможности использования многопоточности, поддержку динамических библиотек, а также включение и исключение отладочных печатей при работе сборщика мусора.

В силу того, что реализация доступа к памяти зависит напрямую от особенностей архитектуры процессора и операционной системы, то существуют три основных решения данной задачи:

1. Получения информации об участках памяти, для которых необходимо совершить процесс очистки, путем передачи явных фиксированных адресов;

2. Использование данных, содержащихся в файлах каталога /proc, в которых описывается информация о процессах, состоянии и конфигурации ядра и системы;

3. Создание файлов для конкретной архитектуры, в которых содержится описание получения доступа к памяти путем программной реализации на языке Ассемблера.

Данные решения не являются универсальными, для большинства платформ используется комбинации нескольких решений, учитывающих особенности архитектуры.

1.6 Особенности микропроцессорной архитектуры SPARC

Масштабируемая процессорная архитектура SPARC (Scalable Processor Architecture) разработана компанией Sun Microsystems в 1986 году. Архитектура SPARC – это архитектура системы команд центрального процессора, полученная на основе понятия о вычислительной машине с ограниченным набором команд RISC (Restricted (Reduced) Instruction Set Computer).

Первоначально архитектура SPARC была разработана с целью упрощения реализации 32-битового процессора и получила название SPARC V7. Она обладала всеми чертами классических RISC процессоров, сочетая простоту набора команд и высокую скорость исполнения кода. Впоследствии, по мере улучшения технологии изготовления интегральных схем, она постепенно развивалась и в 1990 году опубликована спецификация SPARC V8, а затем, в 1993 году – 64-битовая версия архитектуры (SPARC V9), положенная в основу новых микропроцессоров, получивших название UltraSPARC[11].

Процессоры архитектуры SPARC V8 (МЦСТ R150 и МЦСТ R500) – первые продукты отечественной реализации. Благодаря надёжности, стабильной производительности и умеренному энергопотреблению они хорошо зарекомендовали себя в различных управляющих комплексах.

В настоящее время завершена разработка микропроцессора МЦСТ R1000, совместимого с архитектурой SPARC V9. Характеристики R1000 допускают его применение как в системах управления, так и при решении задач обработки данных.

В соответствии со спецификациями SPARCv8 [12, 13] микропроцессор МЦСТ-R500 характеризуется следующими свойствами:

- линейное 32-разрядное адресное пространство;

- небольшое количество простых форматов команд RISC-класса. Все команды 32-разрядные и выровнены в памяти по границе 32-разрядных слов. Имеется всего три базовых формата команд, в которых поля кода операции и

регистровых операндов всегда находятся в одних и тех же разрядах. Доступ к памяти и ввод/вывод могут осуществляться только командами чтения/записи;

- небольшое количество способов адресации. Адрес по памяти вычисляется либо как «регистр + регистр», либо как «регистр + непосредственное значение, литерал»;

- трехадресная регистровая команда — команды большей частью выполняют действия с двумя операндами (двумя регистрами или одним регистром и константой), помещая результат в третий регистр;

- 136-регистровый файл с 8 окнами по 16 регистров и окном из 8 глобальных регистров. В каждый отдельный момент времени программа «видит» 8 глобальных целочисленных регистров, 16-регистровое текущее окно и 8 регистров из окна предыдущей процедуры. Регистровое окно может трактоваться как способ ускоренного доступа к параметрам процедуры, локальным значениям и адресам возврата;

- отдельный регистровый файл вещественных регистров. Файл может трактоваться в программах как набор из 32 регистров одинарного формата (32-разрядных), или 16 регистров двойного формата (64-разрядных), или как смесь тех или иных;

- задержанная передача управления. Процессор всегда выбирает команду, следующую за командой передачи управления. Эта команда может быть выполнена или не выполнена в зависимости от состояния аннулирующего разряда в команде передачи управления;

- быстрые обработчики прерываний. Прерывания собраны в линейную таблицу, их генерация приводит к захвату в регистровом файле нового регистрового окна.

Для каждой процедуры отводится область в памяти, называемая стеком вызова процедур. При определенных условиях возможна оптимизация, когда удается вместо создания собственной области стека использовать область вызвавшей процедуры.

Во время трансляции в области стека каждой процедуры всегда отводится место для 16 слов, нужных для сохранения входных и локальных регистров процедуры в случае выхода за верхнюю границу окон. Кроме того, в общем случае в стеке процедур отводится место:

- для одного слова для передачи скрытого (неявного) параметра. Этот параметр используется, если вызвавшая процедура ожидает получения от вызванной составного объекта в качестве возвращаемого значения. Скрытое слово содержит адрес по памяти в области стека вызвавшей процедуры;

- шести слов, куда вызванная процедура может записывать те параметры, для которых требуется вычислять адреса.

При необходимости в стеке может отводиться память:

- для дополнительных выходных параметров (сверх шести);

- динамических массивов, динамических составных объектов, автоматически размещаемых скаляров, для которых требуется вычислять адреса, и автоматически размещаемых скаляров, которым не находится места на регистрах;

- создаваемых транслятором временных переменных (обычно их слишком много, чтобы транслятору удавалось размещать все их на регистрах);

- вещественных регистров, сохраняемых при вызовах процедур (выполняется, если в процедуре используются команды вещественной арифметики).

Размещение информации в стеке активной процедуры представлено на рисунке 1.1.

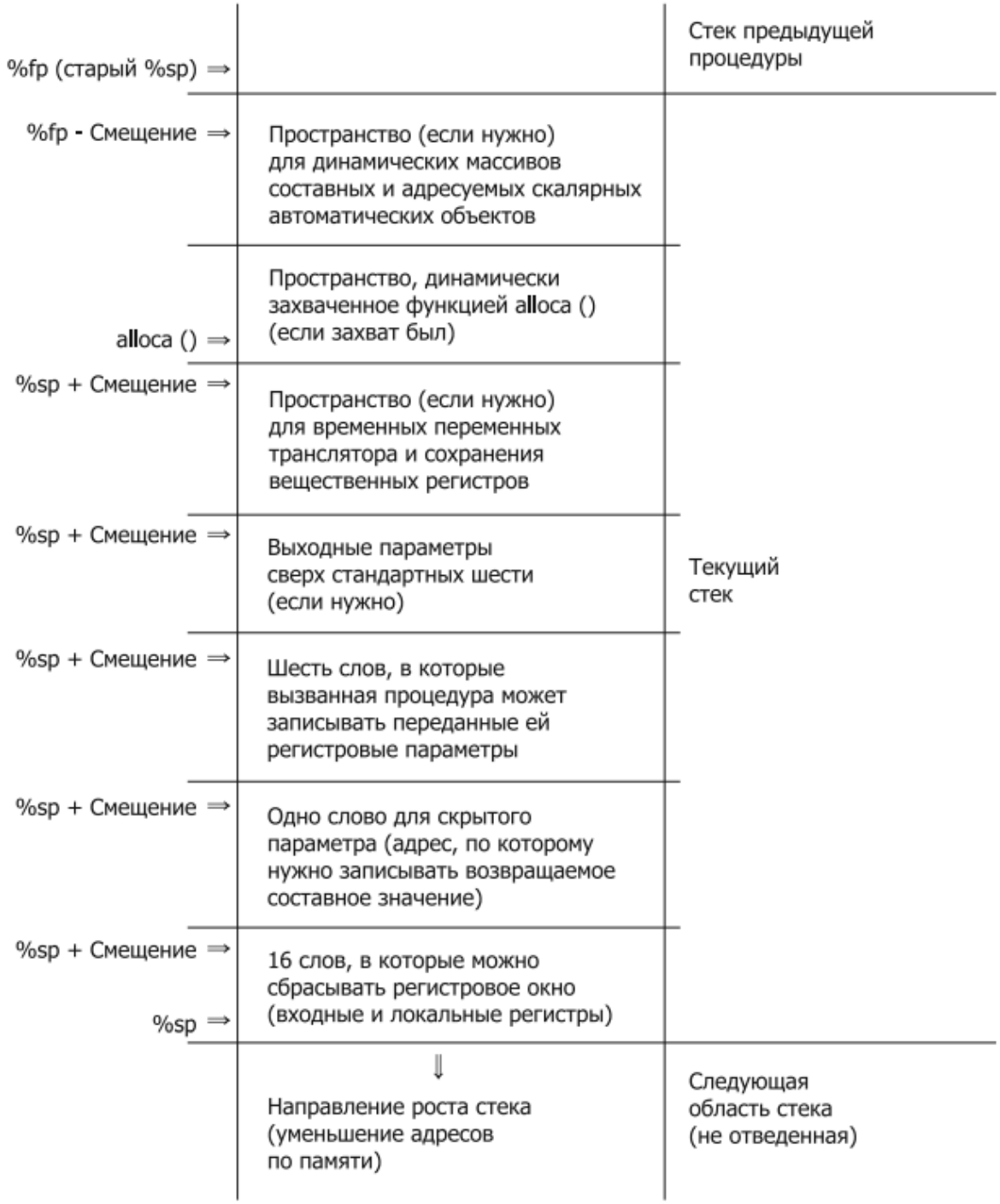

Рисунок 1.1 – Размещение информации в стеке активной процедуры

Динамически (во время выполнения) в стеке может захватываться пространство для памяти, распределяемой с помощью библиотечной функции alloca() языка С. Доступ к адресуемым автоматически переменным, размещенным в стеке, обеспечивается с помощью отрицательных смещений относительно регистра локальной базы %fp.

Динамически распределяемая память адресуется с помощью положительных смещений относительно указателя, полученного от функции alloca().

Все остальные объекты, размещенные в стек, адресуются с помощью положительных смещений относительно регистра указателя магазина %sp.

Глобальные регистры могут использоваться для хранения временных переменных, глобальных переменных и глобальных указателей. Это могут быть и переменные пользователя, и значения, являющиеся частью программного окружения времени выполнения. Регистровое окно охватывает восемь входных и восемь локальных регистров отдельного регистрового набора, а также восемь входных регистров соседнего регистрового набора, которые в текущем регистровом окне адресуются как выходные регистры.

Текущее окно в r-регистрах задается указателем текущего окна CWP, представляющим собой трехразрядное поле счетчика в слове состояния процессора PSR. Указатель текущего окна CWP увеличивается на единицу при выполнении команды восстановления окна вызвавшей процедуры (RESTORE или RETT) и уменьшается на единицу при выполнении команды сохранения окна вызвавшей процедуры (SAVE) или фиксации события.

Выход счетчика за верхнюю и нижнюю границу определяется через регистр маски незначимых окон WIM, управляемый системными программами.

Входные и выходные регистры каждого окна являются общими с двумя соседними окнами. Выходные окна CWP+1 адресуются в текущем окне как входные, выходные текущего окна являются входными для окна CWP-1.

Локальные регистры доступны только в одном (текущем) окне.

Поскольку операции над указателем текущего окна CWP осуществляются по модулю количества окон (NWINDOWS), окно с максимальным номером перекрывается с нулевым окном. Выходные регистры для окна с номером NWINDOWS-1 являются входными для окна с нулевым номером. Окна должны нумероваться непрерывно в диапазоне от 0 до NWINDOWS-1.

Схема перекрытия соседних окон и адресация регистров представлены на рисунке 1.2.

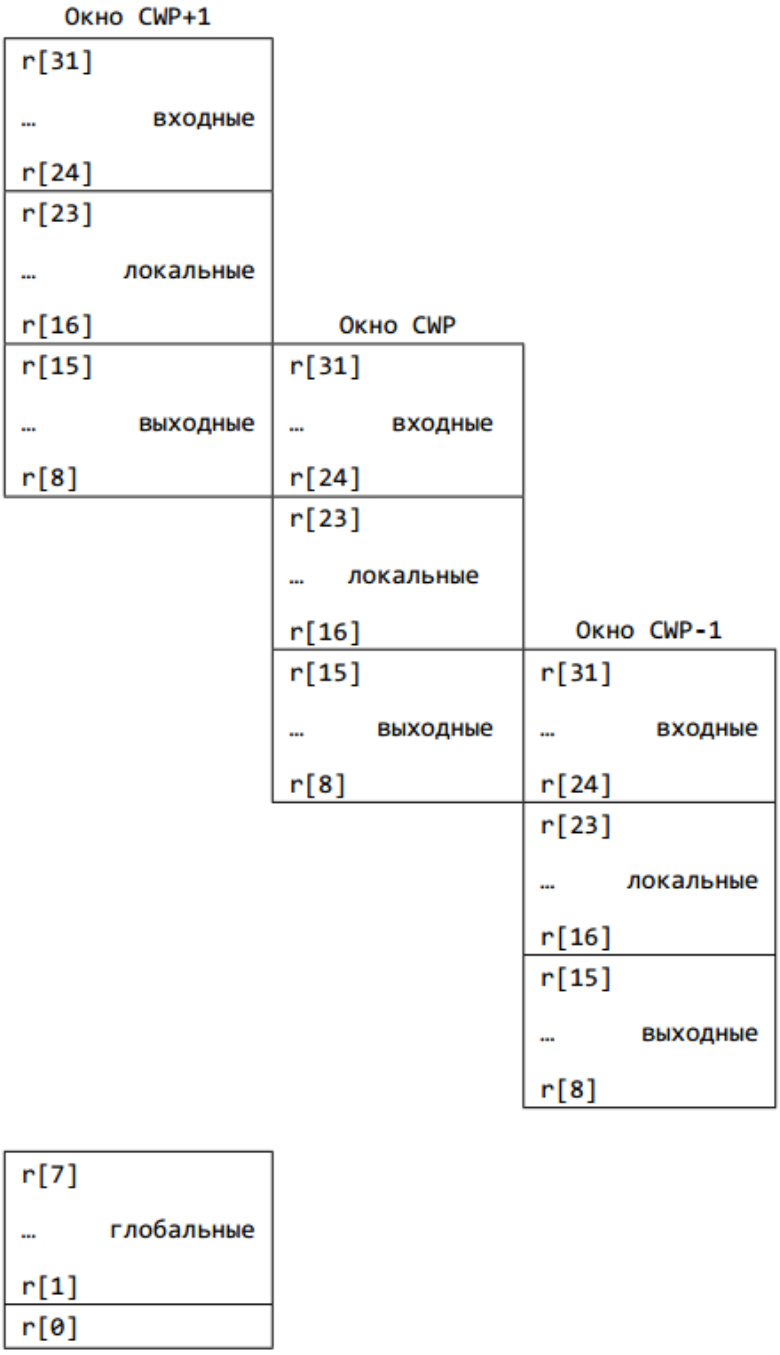

Рисунок 1.2 – Перекрывающиеся окна и глобальные регистры

Реализованная в архитектуре SPARC организация регистровых окон позволяет обеспечить ускоренный доступ к параметрам процедур, локальным значениям и адресам возврата.

1.7 Особенности микропроцессорной архитектуры «Эльбрус»

В качестве важнейшего результата в области разработки отечественных микропроцессоров выделяется создание уникальной микропроцессорной архитектуры «Эльбрус», ориентированной на получение максимальной для данных аппаратных ресурсов показателей производительности [14].

Принципиальной особенностью архитектуры «Эльбрус» является возможность при компиляции каждого фрагмента программы предопределить максимальное распараллеливание вычислительного процесса на всем поле доступных аппаратных ресурсов, которая базируется на использовании широкого командного слова. Соответственно, в общепринятой классификации архитектуру «Эльбрус» можно отнести к категории VLIW (Very Long Instruction Word) [2], [15].

Широкое командное слово, или широкая команда (ШК), содержит набор операций (с их адресными, литеральными и функциональными параметрами), которые одновременно дешифруются и параллельно выполняются, каждая в своем отдельном конвейере. Это принципиальный фактор реализации параллелизма, свойственного данному программному коду. Микропроцессор «Эльбрус» имеет шесть каналов для выполнения арифметико-логических операций. Помимо них в одной ШК могут быть заданы операции и других типов с фиксированными временами выполнения, что позволяет статически, во время трансляции, планировать параллельную работу исполнительных устройств.

Разрядность объектов архитектуры определена в следующих единицах: байт, полуслово (16 разрядов), одинарное слово, или просто слово (32 разряда), двойное слово (64 разряда), квадрослово (128 разрядов).

Широкая команда выровнена по границе двойного слова и имеет длину от 1 до 8 двойных слов. Она состоит из слогов длиной 4 байта и полуслогов длиной 2 байта, которые по отдельности или в комбинациях кодируют операции.

Общая структура микропроцессора «Эльбрус» представлена в виде блок-схемы на рисунке 1.3.

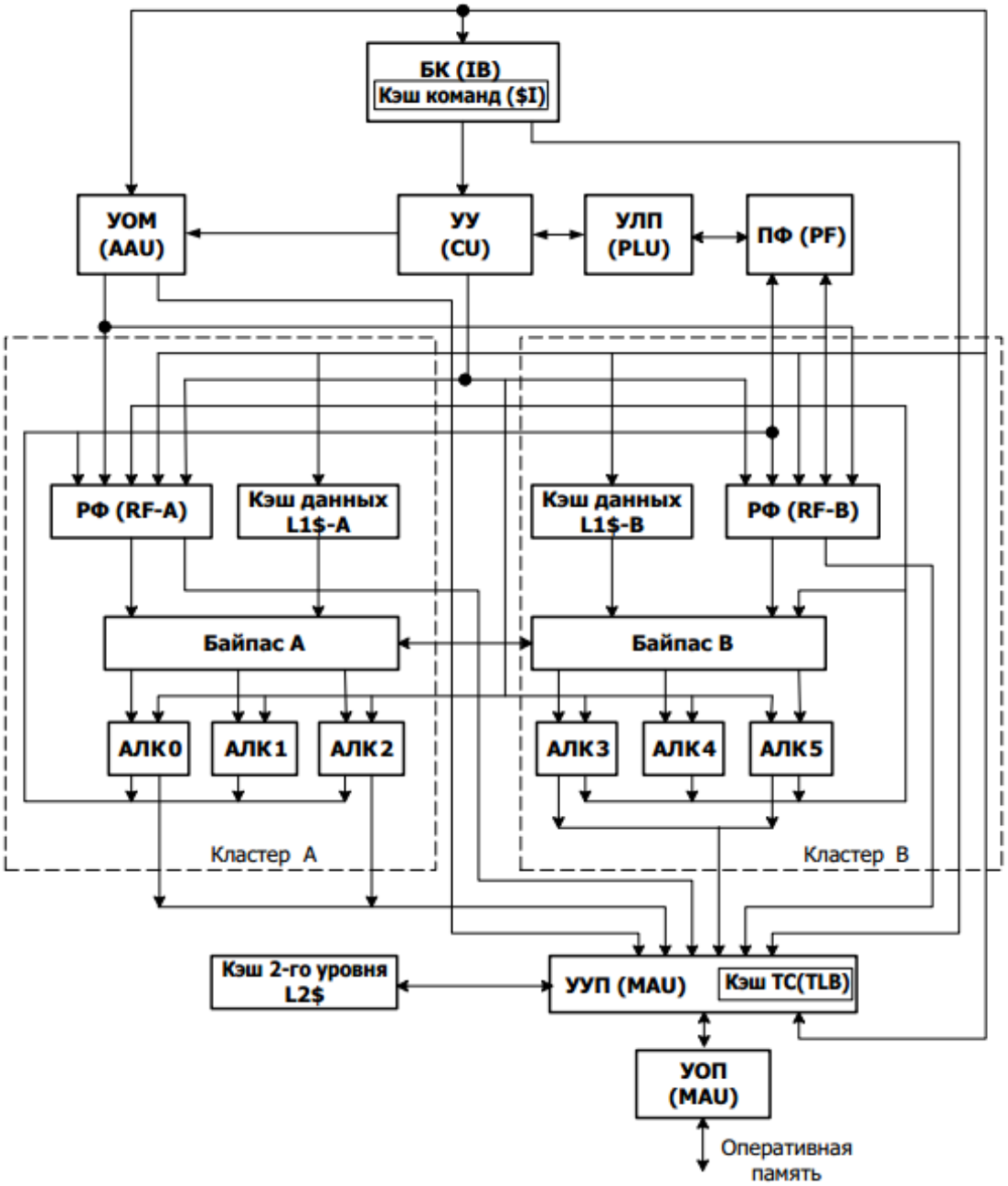

Рисунок 1.3 – Блок-схема микропроцессора «Эльбрус»

БК — буфер команд (IB — Instruction Buffer) предназначен для вызова программного кода из памяти, буферизации на время, достаточное для обеспечения непрерывности дешифрации, и выдачи его в устройство управления для последующей обработки. Накопление программного (объектного) кода обеспечивает кэш команд I$ емкостью 64 Кбайт.

УУ — устройство управления (CU — Control Unit) выполняет считывание программного кода из буферной памяти БК, распаковку широких команд, их дешифрацию и переключение ветвей программы при выполнении команд переходов. С этой целью в структуре УУ предусмотрены обработка последовательности ШК основной ветви, предварительная подкачка ШК трех ветвей предполагаемого ветвления и распаковка первой из команд каждой ветви. Заметим, что подготовка перехода выполняется на фоне выполнения основной ветви и поэтому не приводит к замедлению вычислительного процесса. С выхода УУ в исполнительные устройства выдается распакованная ШК основного потока или одного из подготовленных потоков (при наличии в основном потоке операции передачи управления с реализовавшимся условием перехода).

ПФ — предикатный файл (PF — Predicate File) хранит первичные предикаты — битовые значения, выработанные операциями сравнения, и вторичные предикаты — результаты логических операций над первичными предикатами. Предикатный файл — набор 32 двухразрядных регистров (по одному разряду для предиката и тега). С помощью предикатных значений может быть задан режим условного выполнения операции или условий для команд ветвлений.

УЛП — устройство логических предикатов (PLU — Predicate Logic Unit) предназначено для выполнения операций считывания предикатов из файла ПФ и логических операций формирования вторичных предикатов, которые также могут быть записаны в предикатный файл. Эти операции выполняются над малоформатными значениями, поэтому в целях разгрузки арифметико-логических каналов обработка предикатов возложена на УЛП. Ее результаты направляются в устройство управления для задания режима условного выполнения команд, отрабатываемых в арифметико-логических каналах, или выполнения команд передачи управления.

Арифметико-логические каналы (АЛК) предназначены для исполнения обычных арифметических и логических операций, операций обращения к

памяти и обработки адресных данных (дескрипторов, указателей и др.). В состав микропроцессора входят шесть арифметико-логических каналов (АЛК0—АЛК5), разделенных на два кластера. Арифметико-логические каналы работают параллельно и исполняют в основном одинаковый набор операций. В качестве операндов служат данные из РгФ или результаты других исполнительных устройств. Не все каналы идентичны, поскольку не все операции выполняются одинаково часто. Например, довольно редкая операция деления реализована только в АЛК5. Арифметико-логические каналы имеют отдельные устройства для исполнения целочисленных и вещественных операций. Это не относится к целочисленным операциям умножения и деления — для них введены соответствующие блоки в вещественных исполнительных устройствах.

Регистровый файл (РгФ) предназначен для хранения локальных данных процедуры и результатов выполненных операций. Он представляет собой сверхоперативное запоминающее устройство с произвольным доступом, обращение которому осуществляется через порты. Многочисленность абонентов регистрового файла требует большого количества портов для обслуживания всех запросов одновременно. С целью сокращения их количества регистровый файл реализован в виде двух одинаковых блоков — РгФ-А и РгФ-В, по одному в каждом кластере. Причем, в отличие от используемых во многих микропроцессорах, блок является общим для целочисленных и вещественных устройств арифметики, что позволило повысить эффективность его использования. Блоки регистрового файла содержат одни и те же данные, поскольку результат любой операции записывается одновременно в оба блока. Этим обеспечивается когерентность данных — свойство, которое используется при работе с общими данными.

Блок РгФ содержит 256 регистров. При запуске процедуры ей выделяется определенный участок смежных регистров, называемый окном. Активная (выполняемая в данный момент) процедура может обращаться только к регистрам своего окна. Если все регистры окажутся занятыми, автоматически

выполняется откачка. В результате откачки данные пересылаются в стек процедур, размещенный в оперативной памяти. Таким образом, регистровый файл служит в качестве аппаратной вершины стека процедуры. Все ранее работавшие, но не завершенные процедуры сохраняют свои окна в регистровом файле. Возврат процедур к своим окнам осуществляется по мере окончания работы запущенных ими процедур. Иными словами, реализуется дисциплина (LIFO) — активизируется та процедура, которая запустила завершившуюся в данный момент времени процедуру.

Чтобы результат, полученный некоторым исполнительным устройством, мог быть использован другим до записи в РгФ, исполнительные устройства связаны шинами байпаса. Благодаря этому в ряде случаев отпадает необходимость в считывании операндов из РгФ. Каждый кластер имеет отдельный блок байпаса, с помощью которого результаты передаются исполнительным устройствам не только своего, но и другого кластера. Кроме того, байпас позволяет использовать результаты считывания из кэш-памяти L1$ до их записи в регистровый файл.

*Кэш данных первого уровня L1$* выполнен в виде двух одинаковых блоков (L1$-A и L1$-B) емкостью 64 Кбайт, по одному в каждом кластере. Блоки L1$ хранят одинаковые данные, поскольку запись данных выполняется одновременно в оба блока. В блоке хранятся данные, которые используются в качестве операндов для исполнительных устройств АЛК. Но поскольку в общем случае операнды считываются из регистрового файла, они должны быть предварительно загружены в него из кэша L1$. Для этих целей предусмотрены операции загрузки, выполняемые исполнительными устройствами АЛК. В случае отсутствия требуемых данных в кэше L1$ операция загрузки продолжается поиском данных в кэш-памяти второго уровня L2$ с последующей записью их в оба блока L1$. Параллельная запись также, как и в случае регистрового файла, вызвана необходимостью сохранения свойства когерентности данных обоих блоков L1$.

*Кэш второго уровня L2$* является общим для данных и программного кода, его объем составляет 256 Кбайт, степень ассоциативности — 4. Обращение к L2$ выполняется при отсутствии требуемых данных в L1$ или нужного программного кода в буферной памяти команд устройства БК. Если нужная информация отсутствует и в L2$, то формируется запрос к оперативной памяти. Считанная из ОП информация поступает потребителю и одновременно записывается в кэш-память вместо устаревших данных.

УОМ — Устройство обращения к массивам (AAU — Array Access Unit) предназначено для упреждающей подкачки элементов массива при выполнении векторных операций. Поскольку каждый элемент вектора обрабатывается одной и той же последовательностью операций, то обработка идет циклически. В микропроцессоре «Эльбрус» к началу очередного цикла нужный элемент вектора уже считан из памяти (кэша L2$ или ОП) и находится в буфере УОМ. Подкачка элементов массивов в буфер, использующий дисциплину очереди (FIFO), осуществляется на фоне выполнения основной (синхронной) программы параллельной ей (асинхронной) программой.

УУП — устройство управления памятью (MMU — Memory Management Unit) преобразует виртуальные адреса в физические. С целью ускорения этого процесса наиболее часто используемые строки таблицы страниц хранятся в кэше таблицы страниц (TLB) объемом 64 строки. Если нужная строка в нем отсутствует, выполняется аппаратный поиск в таблице страниц, хранящейся в памяти, и загрузка найденной строки в кэш вместо устаревшей. Кроме того, рассматриваемое устройство обеспечивает обращения в кэш L2$ и ОП.

УОП — устройство обращения в память (MAU — Memory Access Unit) предназначено для связи микропроцессора с ОП. Оно содержит буферы операций считывания и записи, позволяющих осуществить потоковое обслуживание заявок. Обмен с памятью осуществляется через 16-байтовый канал с раздельными шинами для передачи и приема данных. Обмен выполняется блоками по 32 или 64 байт.

2 Постановка задачи

2.1 Постановка задачи организации доступа к памяти для архитектуры микропроцессора «Эльбрус»

Имеется реализация процесса сборки мусора для языков программирования C/C++, разработанная Хансом Боэмом. Необходимо реализовать сборщик мусора для архитектуры микропроцессора «Эльбрус».

В силу того, что архитектурно-зависимая часть при портировании технологии автоматической очистки оперативной памяти на другую систему требует не только переопределения данных в конфигурационных файлах, но и реализации доступа к памяти с учетом всех особенностей архитектуры системы, необходимо осуществить доступ ко всем участкам памяти необходимым для корректной работы сборщика мусора.

Для решения поставленной задачи необходимо изучить возможные аналоги, реализованные для процессоров со схожей архитектурой; изучить принцип организации памяти для выполняемых процессов; на примере существующих реализаций разработать решение на языке Ассемблера для микропроцессора «Эльбрус», а также учесть особенности компилятора и его уровней оптимизации.

2.2 Тестирование реализации алгоритма и анализ результатов

Для проверки работоспособности разработанного решения необходимо провести внутренние тесты сборщика мусора, а также провести анализ полученных результатов путем сравнения с реализациями для архитектур других процессоров.

В ходе проведения тестирований необходимо использовать инструменты для профилирования скорости и производительности выполняемых

исследований, а также инструменты для отладки и дальнейшей оптимизации решения.

Анализ результатов необходимо проводить с учетом особенностей конкретно взятых систем, а также, используя профилировщик, отслеживать возможные недочеты реализации.

2.3 Кроссплатформенная сборка

По завершению портирования сборщика мусора под архитектуру микропроцессора «Эльбрус» необходимо реализовать кроссплатформенную сборку для таких систем как: SPARC, x86, E2K.

Для осуществления кроссплатформенной сборки необходимо ознакомиться с системой кросс-компиляции, особенностями создания сценариев сборки, а также архитектурно-зависимой реализацией сборщика мусора для каждой системы.

По выполнению поставленной задачи необходимо протестировать проделанную работу путем установки готового пакета на машины, поддерживающие каждую из архитектур. Помимо этого, необходимо провести внутренние тесты сборщика мусора для обнаружения возможных ошибок при реализации процесса.

2.4 Оптимизация решения

В системе микропроцессоров «Эльбрус» процесс сборки мусора необходим для использования сторонних приложений и программ, которые в своем составе используют библиотеки сборщика мусора. Для корректной работы таких программ необходима грамотная очистка памяти. При эксплуатации утилит, использующих библиотеки сборщика мусора, могут выявиться недочеты и ошибки, допущенные при портировании процесса сборки мусора под архитектуру микропроцессоров «Эльбрус».

Одним из продуктов программного обеспечения, использующего библиотеки реализуемого сборщика мусора, является реализация языка программирования Scheme – Guile. Данный язык программирования не имеет личного сборщика мусора, а для решения проблем с очисткой памяти использует библиотеки, реализованные для языков программирования C/C++.

На примере тестирования работы guile, использующего реализованный процесс сборки мусора, необходимо выявить возможные ошибки и недостатки и устранить их.

3 Изучение реализованных методик решений

Проект сборки мусора, реализованный Хансом Боэмом, не до конца портирован под все системы и архитектуры, но на основе уже существующих решений возможно создать собственную реализацию, которая будет учитывать все особенности доступа к памяти.

В качестве исследуемых методик решений были выбраны архитектуры микропроцессоров семейства Intel, таких как: x86 и IA-64, а также архитектура RISC-микропроцессоров – SPARC. Выбор данных реализаций был обусловлен следующими причинами:

- процесс сборки мусора полностью реализован под архитектуру микропроцессора x86. В настоящее время данное решение является наиболее распространённым и используемым. Для выполнения поставленной задачи о кроссплатформенной сборке garbage collector-а необходима также реализация под архитектуру x86;

- в общей классификации IA-64, как и микропроцессоры семейства «Эльбрус», относится к категории архитектур, использующих принцип широкого командного слова (Very Large Instruction Word, VLIW), когда компилятор формирует для параллельного исполнения последовательности групп команд (широкие командные слова), в которых отсутствуют зависимости между командами внутри группы и сведены к минимуму зависимости между командами в разных группах;

- для архитектур IA-64 и SPARC в проекте сборки мусора реализован доступ к участкам памяти при помощи архитектурно-зависимых файлов, в которых содержится описание получения доступа к памяти путем программной реализации на языке Ассемблера. По такому принципу необходима реализация для архитектур микропроцессоров «Эльбрус»;

- в семействе микропроцессоров «Эльбрус» существуют ряд процессоров на базе архитектуры SPARC. Кроссплатформенная сборка подразумевает портирование и на данную архитектуру.

3.1 Описание существующих аналогов

В архитектуре микропроцессоров x86 поддерживается стек, а именно непрерывная область оперативной памяти, представляющей собой список элементов, организованных по принципу LIFO (англ. last in – first out, «последним пришёл – первым вышел»). Для работы со стеком в процессоре реализованы специальные машинные коды, ассемблерные мнемоники.

Стек представляется как одномерный массив с упорядоченными адресами. Такая организация стека удобна, если элемент информации занимает в памяти фиксированное количество слов, например, одно слово. При этом отпадает необходимость хранения в элементе стека явного указателя на следующий элемент стека, что экономит память. При этом указатель стека (Stack Pointer – SP) является *регистром процессора* и указывает на адрес головы стека.

Стек в памяти растет сверху вниз, это значит, что при добавлении значения в него адрес стека уменьшается, а при извлечении адрес вершины стека увеличивается. Таким образом, при каждом вталкивании слова в стек, SP сначала уменьшается на один и затем по адресу из SP производится запись в память, а при каждом выталкивании слова из стека сначала производится чтение по текущему адресу из SP и последующее увеличение содержимого SP на 1.

С точки зрения организация памяти важную роль играют *регистры процессора* — блок ячеек памяти, образующий сверхбыструю оперативную память (СОЗУ) внутри процессора; используется самим процессором и большей частью недоступен программисту: например, при выборке из памяти очередной команды она помещается в регистр команд, к которому программист обратиться не может.

Имеются также регистры, которые, в принципе, программно доступны, но обращение к ним осуществляется из программ операционной системы,

например, *управляющие регистры* и *теневые регистры дескрипторов сегментов*. Этими регистрами пользуются в основном разработчики операционных систем.

Существуют также так называемые *регистры общего назначения* (РОН), представляющие собой часть регистров процессора, использующихся без ограничения в арифметических операциях, но имеющие определенные ограничения, например, в строковых РОН, не характерные для эпохи мейнфреймов типа IBM/370 стали популярными в микропроцессорах архитектуры X86 — Intel 8085, Intel 8086 и последующих[17].

Специальные регистры содержат данные, необходимые для работы процессора — смещения базовых таблиц, уровни доступа и т. д. Часть специальных регистров принадлежит устройству управления, которое управляет процессором путём генерации последовательности микрокоманд.

Доступ к значениям, хранящимся в регистрах, как правило, в несколько раз быстрее, чем доступ к ячейкам оперативной памяти (даже если кеш-память содержит нужные данные), но объём оперативной памяти намного превосходит суммарный объём регистров (объём среднего модуля оперативной памяти сегодня составляет 1—4 гигабайт, суммарная «ёмкость» регистров общего назначения/данных для x86-процессоров, например Intel 80386 и более новых, 8 регистров по 4 байта = 32 байта; в режиме x86 64 - 16 по 8 байт = 128 байт и некоторое количество векторных регистров).

В процессоре есть несколько наборов логик, каждая из которых имеет свои машинные коды и свои наборы регистров.

Основные программные регистры (Basic Program register – BP). Эти регистры используются всеми программами, с их помощью выполняется обработка целочисленных данных. В архитектуре микропроцессоров x86 для 32-разрядной системы предусмотрено 8 целочисленных регистров, а для 64-разрядной системы 16.

Для работы с данными представленными в формате с плавающей точкой также реализованы регистры (Floating Point Unit registers – FPU). В x86 32-

разрядной системе отведено 8 FPU регистров, а в 64-разрядной системе отведено 16.

Для выполнения одной инструкции над большим количеством операндов существуют MMX и XMM регистры.

На примере архитектуры x86 рассмотрим основные группы регистров:

IP (англ. Instruction Pointer) — регистр, указывающий на смещение (адрес) инструкций в сегменте кода (1234:0100h сегмент/смещение). IP — 16-битный (младшая часть EIP), EIP — 32-битный аналог (младшая часть RIP), RIP — 64-битный аналог [8].

Сегментные регистры — регистры, указывающие на сегменты. Все сегментные регистры - 16-разрядные. CS (англ. Code Segment), DS (англ. Data Segment), SS (англ. Stack Segment), ES (англ. Extra Segment), FS, GS.

В реальном режиме работы процессора сегментные регистры содержат адрес начала 64Kb сегмента, смещенный вправо на 4 бита. В защищенном режиме работы процессора сегментные регистры содержат селектор сегмента памяти, выделенного ОС.

CS — указатель на кодовый сегмент. Связка CS:IP (CS:EIP/CS:RIP — в защищенном/64-битном режиме) указывает на адрес в памяти следующей команды. В 64-разрядном режиме сегментные регистры CS, DS, ES и SS в формировании линейного (непрерывного) адреса не участвуют, поскольку сегментация в этом режиме не поддерживаются.

Регистры данных применяются для хранения промежуточных данных, участвуют в арифметических и логических операциях процессора. Аккумулятор, базовые и индексные регистры также можно использовать в качестве регистров данных.

RAX, RCX, RDX, RBX, RSP, RBP, RSI, RDI, R8 — R15 — 64-битные.

EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI, R8D — R15D — 32-битные (extended AX).

AX (англ. Accumulator), CX (англ. Count Register), DX (англ. Data Register), BX (англ. Base Register), SP (англ. Stack Pointer), BP (англ. Base

Pointer), SI (англ. Source Index), DI (англ. Destination Index), R8W — R15W — 16-битные.

AH, AL, CH, CL, DH, DL, BH, BL, SPL, BPL, SIL, DIL, R8B — R15B — 8-битные (половинки 16-битных регистров), например, AH — high AX — старшая половинка 8 бит, AL — low AX — младшая половинка 8 бит.

Общая структура регистров данных представлена на рисунке 3.1.

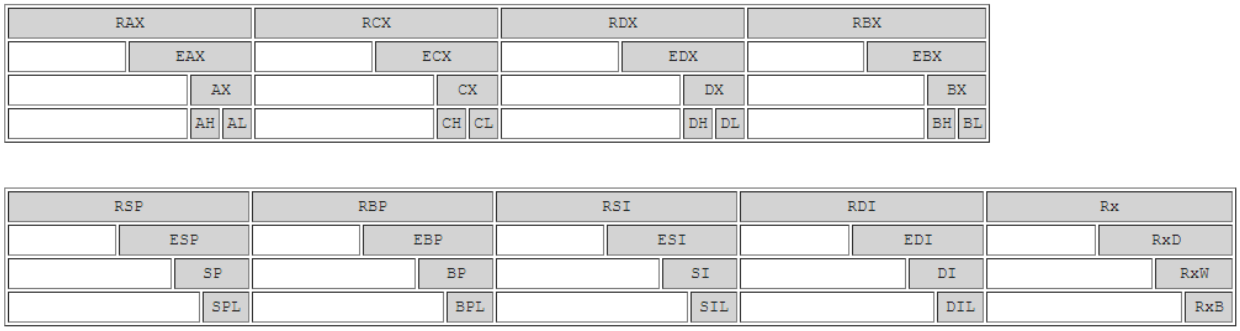

Рисунок 3.1 – Структура регистров данных

Аккумулятор (англ. Accumulator) – это регистр, который применяется для хранения промежуточных результатов вычисления. Именно его предпочтительно выбирать в качестве хранения результата действия арифметической и логической операции. Он содержит первый множитель или делимое в операциях умножения/деления. при этом многие операции с участием этого регистра (не обязательно умножение и деление) часто выполняется быстрее [15]. Он называется "аккумулятором" потому, что он как бы "накапливает" (аккумулирует) результаты вычислений перед их посылкой в память компьютера.

Счетчик повторений (англ. Counter). Этот регистр участвует в качестве счетчика для некоторых команд (сдвига, манипулирования со строками, цепочками, организации циклов и повторений.) [6], [17]

Индексные регистры (Index register) участвуют в операциях с индексной адресацией. Под этими операциями понимают те, в которых адрес элемента в памяти вычисляется, например, по правилу: <база>+<содержимое индексного регистра>*<длина элемента данных>. Эта адресация используется при

операциях с массивами, строками и другими линейно-упорядоченными объектами, то есть объектами однотипных данных, расположенных в памяти "подряд". При некоторых операциях различают *индекс источника* (Source Index) и *индекс получателя*, или приемника (Destination Index). Примером могут служить циклические операции со строками.

Регистры RAX, RCX, RDX, RBX, RSP, RBP, RSI, RDI, Rx, RxD, RxW, RxB, SPL, BPL, SIL, DIL доступны только в 64-битном режиме работы процессора.

Регистр флагов FLAGS (16 бит) / EFLAGS (32 бита) / RFLAGS (64 бита) — содержит текущее состояние процессора. Данный тип регистров создает признак результата необходимый в операциях ветвления и циклов, поэтому происходит наиболее частое обращение у программистов.

Системные регистры GDTR, LDTR и IDTR введены в процессорах начиная с Intel286 и предназначены для хранения базовых адресов таблиц дескрипторов — важнейших составляющих системной архитектуры при работе в защищенном режиме.

Регистр GDTR содержит 32-битный (24-битный для Intel286) базовый адрес и 16-битный предел глобальной таблицы дескрипторов (GDT).

Видимая часть регистра LDTR содержит только селектор дескриптора локальной таблицы дескрипторов (LDT). Сам дескриптор LDT автоматически загружается в скрытую часть LDTR из глобальной таблицы дескрипторов.

Регистр IDTR содержит 32-битный (24-битный для Intel286) базовый адрес и 16-битный предел таблицы дескрипторов прерываний (IDT). В реальном режиме может быть использован для изменения местоположения таблицы векторов прерываний.

Видимая часть регистра TR содержит селектор дескриптора сегмента состояния задачи (TSS). Сам дескриптор TSS автоматически загружается в скрытую часть TR из глобальной таблицы дескрипторов.

Регистром называется функциональный узел, осуществляющий приём, хранение и передачу информации. Регистры состоят из группы триггеров. По типу приёма и выдачи информации различают 2 типа регистров:

- с последовательным приёмом и выдачей информации — *сдвиговые регистры*;

- с параллельным приёмом и выдачей информации — *параллельные регистры*.

Сдвиговые регистры представляют собой последовательно соединённую цепочку триггеров. Основной режим работы — сдвиг разрядов кода от одного триггера к другому на каждый импульс тактового сигнала.

По назначению регистры различаются на:

- *аккумулятор* — используется для хранения промежуточных результатов арифметических и логических операций и инструкций ввода-вывода;

- *флаговые* — хранят признаки результатов арифметических и логических операций;

- *общего назначения* — хранят операнды арифметических и логических выражений, индексы и адреса;

- *индексные* — хранят индексы исходных и целевых элементов массива;

- *указательные* — хранят указатели на специальные области памяти (указатель текущей операции, указатель базы, указатель стека);

- *сегментные* — хранят адреса и селекторы сегментов памяти;

- *управляющие* — хранят информацию, управляющую состоянием процессора, а также адреса системных таблиц.

При преобразовании логической памяти в физическую есть промежуточный этап:

Логический адрес – Линейный (виртуальный) адрес – Физический адрес

Все линейное адресное пространство разбито на сегменты. Адресное пространство каждого процесса имеет по крайней мере три сегмента:

- *Сегмент кода* – содержит команды из программы, которые будут исполняться;

- *Сегмент данных* – содержит данные, а именно переменные;

- *Сегмент стека*;

Общая схема организация линейного пространства представлена на рисунке 3.2:

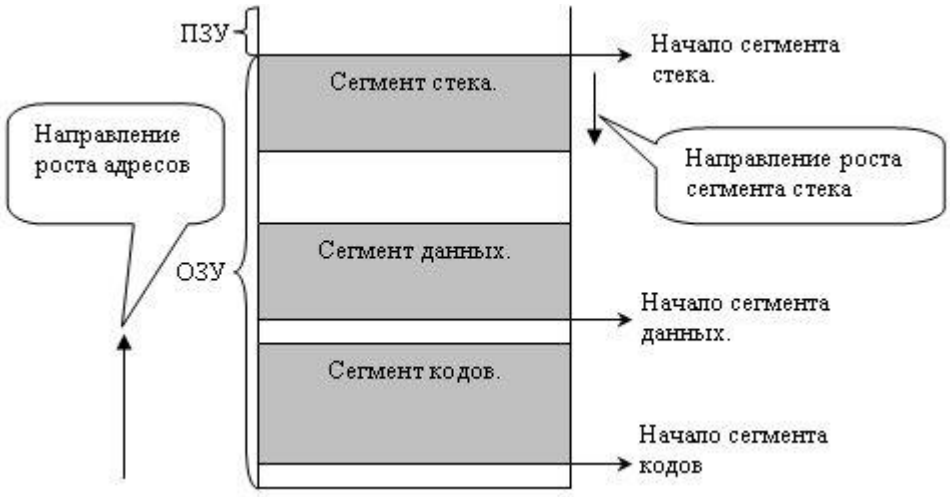

Рисунок 3.2 – Организация линейного пространства

Линейный адрес вычисляется по формуле:

Линейный адрес = Базовый адрес + Смещение;

Сегмент кода.

Базовый адрес сегмента кода берется из регистра CS. Значение смещения для сегмента кода берется из регистра EIP, в котором хранится адрес инструкции, после исполнения которой, значение EIP увеличивается на размер этой команды. Если команда занимает 4 байта, то значение EIP увеличивается на 4 байта и будет указывать уже на следующую инструкцию. Все это делается автоматически без участия программиста. Сегментов кода может быть несколько в памяти. В случае 32-разрядной системы сегмент кода один.

Сегмент данных.

Данные загружаются в регистры DS, ES, FS, GS. Это означает, что сегментов данных может быть до 4х. Смещение внутри сегмента данных задается как операнд команды. По умолчанию используется сегмент, на

который указывает регистр DS. Для того, чтобы войти в другой сегмент, необходимо указать явным образом в команде префикса замены сегмента.

Сегмент стека.

Используемый сегмент стека задается значением регистра SS. Смещение внутри этого сегмента представлено регистром ESP, который указывает на вершину стека. Сегменты в памяти могут друг друга перекрывать, мало того базовый адрес всех сегментов может совпадать, например, в нуле. Такой вырожденный случай называется *линейным представлением памяти*. В современных системах память, как правило, организована по такому принципу.

В регистрах SS, DS, CS содержится 16-битный *селектор*, который указывает на дескриптор сегментов, в котором уже хранится необходимый адрес.

Селектор — число, хранящееся в сегментном регистре; это 16-битная структура данных, которая является идентификатором сегмента. В реальном режиме содержимое каждого сегментного регистра представляет собой номер параграфа — 16-байтового участка памяти, выровненного на границу 16 байт.

В защищённом режиме каждый сегментный регистр делится на три части, как показано на рисунке 3.3:

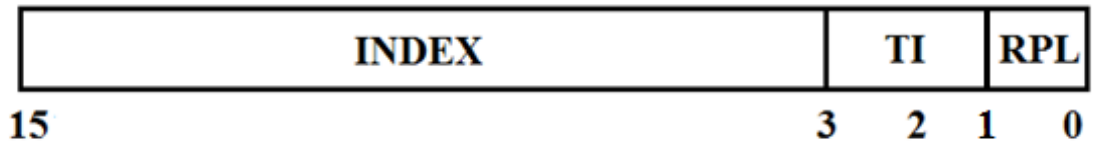

Рисунок 3.3 – Структура селектора

Бит TI в этом случае указывает, какая таблица дескрипторов должна использоваться (нуль соответствует глобальной таблице дескрипторов, единица — локальной таблице дескрипторов).

Поле RPL используется для контроля прав доступа программы к сегменту и является запрошенным уровнем привилегий. Частным случаем RPL является текущий уровень привилегий — CPL, чьё значение в любой момент времени находится в сегментном регистре CS.

В тринадцати битах селектора содержится индекс *дескриптора* в *таблице дескрипторов*, который должен использоваться при вычислении линейного адреса. Максимальное количество дескрипторов в таблице: 2^{13}= 8192.

В архитектуре x86 *дескриптор* представляет собой служебную структуру в памяти, которая определяет сегмент. Длина дескриптора равна 8 байт.

Общая структура дескриптора представлена на рисунке 3.4:

Рисунок 3.4 – Структура сегментного дескриптора

- Базовый адрес (жёлтые поля, 32 бита) — начало сегмента в линейной памяти. При помощи формулы нахождения линейного адреса процессор может обращаться по нужному адресу линейной памяти;

- Предел сегмента (красные поля, 20 бит) — задает размер сегмента. Для сегментов данных и кода доступными являются все адреса, расположенные в интервале [база; база+предел);

- Права доступа (синие поля, 12 бит) — флаги, определяющие наличие сегмента в памяти, уровень защиты, тип, разрядность + один пользовательский флаг.

Байт прав доступа (AR, англ. Access Rights, биты 8-15 на рисунке 3.4):

- Бит P определяет доступность сегмента (0 — сегмента нет, 1 — есть). При обращении к сегменту со сброшенным битом P происходит исключение #NP, обработчик которого может загрузить/создать сегмент.

- Номер привилегий DPL содержит 2-битный номер (0-3), определяющий, к какому уровню (кольцу) защиты относится этот сегмент.

- Тип сегмента (биты 8-12 на рисунке). Старший бит (S) определяет сегмент как системный (S=0) или пользовательский (S=1).

Дескрипторные таблицы — служебные структуры данных, содержащие дескрипторы сегментов.

В архитектуре x86 есть три вида дескрипторных таблиц:

- глобальная дескрипторная таблица (англ. Global Descriptor Table, GDT);

- локальная дескрипторная таблица (англ. Local Descriptor Table, LDT);

- таблица векторов прерываний (англ. Interrupt Descriptor Table, IDT);

Глобальная дескрипторная таблица является общей для всех процессов. Её размер и расположение в физической памяти определяются регистром GDTR. Размер таблицы не может превышать 8192 дескрипторов, поскольку один дескриптор занимает 8 байт, а лимит в регистре GDTR — двухбайтный и хранит размер таблицы минус один (максимальное значение лимита — 65535), а 8192 x 8 = 65536.

Дескрипторы LDT и сегментов задач (TSS) могут находиться только здесь.

Особенностью GDT является то, что у неё запрещён доступ к первому (с нулевым смещением относительно начала таблицы) дескриптору. Обращение к нему вызывает исключение #GP, что предотвращает обращение к памяти с использованием незагруженного сегментного регистра.

Локальная дескрипторная таблица. В отличие от GDT, LDT может быть много (соответственно количеству задач (потоков), но не обязательно). Каждая задача может иметь свою. На расположение таблицы текущей задачи указывает регистр LDTR.

Размер и расположение LDT в линейной памяти определяются дескриптором LDT из GDT (но это не означает, что размер LDT может быть больше 65536 байт). Первый дескриптор LDT можно использовать.

Таблица дескрипторов прерываний. Таблица прерываний глобальна, размещение в физической памяти определяется регистром IDTR.

При возникновении прерывания (внешнего, аппаратного, или вызванного инструкцией Int):

- из IDT выбирается дескриптор шлюза, соответственно номеру прерывания;

- проверяются условия защиты (права доступа);

- при соблюдении условий защиты выполняется переход на подпрограмму-обработчик этого прерывания.

На примере архитектуры x86 описан принцип организации памяти, а именно преобразование логической памяти в физическую.

Для грамотной работы автоматического процесса очистки памяти необходимо осуществить передачу данных о регистрах и стеках сборщику мусора.

3.2 Особенности организации памяти микропроцессора «Эльбрус»

Для осуществления процесса очистки мусора необходимо иметь представление об организации оперативной памяти архитектуры микропроцессора.

В архитектуре «Эльбрус» различают следующие разновидности стеков:

- стек процедур;

- стек пользователя;

- стек связующей информации.

Стек процедур. Стек процедур представляет собой непрерывную область, предназначенную для временного хранения тех фактических параметров и локальных данных еще не завершенных процедур, которые размещены компилятором в операционных регистрах (регистровых окнах). К этой категории данных относятся также результаты выполненных операций, записываемых в регистровый файл.

Для каждой запускаемой процедуры в регистровом файле отводится окно (фрейм) требуемого размера, в пределах которого разрешена ее работа. Окно новой процедуры и предыдущее окно могут иметь общую область, в которой содержатся параметры, передаваемые запускающей процедурой, и

возвращаемые значения — результаты ее работы. Завершение процедуры приводит к ликвидации ее окна.

По мере запуска новых процедур ресурс свободных регистров РгФ может быть недостаточным для очередного окна. В этом случае нижняя часть стека автоматически откачивается в память. И наоборот, после завершения запущенной процедуры и возврата к прерванной процедуре необходимые данные автоматически подкачиваются из памяти в РгФ. В силу этого стек процедур состоит из двух частей, одна из которых располагается в регистровом файле, а для другой отводится участок памяти на случай переполнения РгФ.

Стек пользователя. Стек пользователя предназначен для данных, не относящихся к рассмотренной ранее категории, например, различного рода динамических массивов или объектов программ на объектно-ориентированных языках.

В принципе вопросы выделения памяти под различные структуры данных решаются с помощью стандартных процедур управления динамической памятью типа new/delete. Но обращение к ним связано с дополнительными затратами времени при вызове процедур операционной системы. Поэтому в микропроцессоре реализован более эффективный способ управления динамической памятью, основанный на организации стека для этих целей.

В систему команд введена специальная операция, формирующая дескриптор на участок памяти, выделенный из стека пользователя, вследствие чего процедура может эффективно заказывать себе память. Освобождение памяти выполняется автоматически при возврате из процедуры.

Стек связующей информации. Стек связующей информации предназначен для временного хранения данных о запускающей процедуре, обеспечивающих возврат к ней после отработки запускаемой процедуры. При защищенном программировании пользователь не должен иметь возможности изменять эту информацию, поэтому стек доступен только операционной системе и аппаратуре. Принцип организации стека связующей информации аналогичен стеку процедур.

С точки зрения управления памятью для осуществления процесса сборки мусора важную роль играет работа с регистровым файлом.

Регистровый файл (Register File, RF) предназначен для хранения операндов и результатов выполненных операций, которые в общем случае используются в последующих командах. По уровню иерархии запоминающих устройств РгФ является сверхоперативным запоминающим устройством (СОЗУ) процессора, соответственно, в системе команд предусмотрены операции, реализующие его связь с оперативной памятью вычислительной системы, которые выполняются в арифметико-логических каналах.

Считывание операндов осуществляется на стадии СчО (R) конвейера. Возможность произвести считывание по нескольким адресам реализована за счет разработки многопортового запоминающего устройства. Порт — это узел, обеспечивающий доступ абонента к требуемому регистру файла, который содержит управляющее оборудование и усилители чтения/записи. Все порты жестко закреплены за устройствами: один из них назначается для считывания операнда по первому адресу, к примеру, нулевого АЛК, по другому порту доставляется второй операнд для этого же канала и т. д. В широкой команде в общем случае может быть до 20 адресов операндов, поэтому требуется такое же количество портов чтения. Очевидно, что увеличение количества портов приводит к возрастанию объема оборудования, причем появляется необходимость в дополнительных устройствах, исключающих конфликты при отработке в разных портах одинаковых адресов. Напротив, снижение числа портов позволяет уменьшить площадь кристалла и увеличить тактовую частоту работы регистрового файла. По этой причине РгФ содержит два одинаковых блока RF-A и RF-B для кластеров A и B соответственно. Для поддержания когерентности каждый блок доступен АЛК другого кластера по записи, он имеет 10 портов чтения и 10 портов записи.

Порты чтения обеспечивают считывание операндов для АЛК, причем каждому каналу выделено три порта, что обеспечивает возможность доставки данных для комбинированных операций, требующих трех операндов. Один из

портов чтения предназначен для считывания из РгФ в АЛК2 (АЛК5) операнда для команды записи в память. Через шесть портов записи в РгФ направляются результаты операций АЛК и данные из кэша первого уровня L1$. Четыре порта выделены для записи в РгФ данных, поступающих по шине кэша L2$, и элементов массива из буфера предварительной подкачки. Запись данных в целях их когерентности осуществляется одновременно в оба блока РгФ (таким образом, оба блока РгФ содержат одинаковые данные).

Следует отметить, что уменьшение числа портов в блоке РгФ с 20 до 10 позволило увеличить вдвое тактовую частоту его работы. В результате появилась возможность совместить порты записи и чтения: полтакта процессора занимает режим считывания из РгФ, в другой половине такта выполняется запись.

Память блока РгФ включает 256 регистров по 84 разряда каждый, имеющих три поля. Первые два поля предназначены для хранения 32-разрядных слов с 2-разрядными тегами, двойного слова Ф64, мантиссы вещественного числа, а третье поле отводится для хранения 16-разрядного значения порядка.

В управление регистровым файлом введено три механизма, обеспечивающих корректное и опережающее использование результатов операций в качестве операндов для последующих операций. Все механизмы задействуют адреса, по которым записываются результаты выполненных операций. Эти адреса сохраняются в АЛК и сопровождают операции по мере продвижения их по конвейеру вплоть до записи результата в РгФ.

Первый механизм (Scoreboarding — схема анализа блокировок выдачи ШК из УУ) блокирует очередную широкую команду, если она содержит обращения за еще не готовыми операндами, которые являются результатами предшествующих операций.

Второй механизм обеспечивает управление шинами байпаса, ускоряющими доставку операндов. Его работа основана на сравнении адресов

операндов из ШК с адресами результатов выполненных операций, доступных по шинам байпаса.

Третий механизм предназначен для разрешения так называемой проблемы зависимости по выходу (output dependence). Она возникает, когда две последовательные операции имеют один и тот же адрес результата, но разное время выполнения и первая операция формирует свой результат уже после завершения выполнения второй операции.

Принцип использования. Окно, совокупность смежных регистров РгФ, выделяется для каждой запускаемой процедуры. База окна и его размер задаются компилятором. Окно описывается дескриптором, который хранится на регистре WD (current Window Descriptor).

Все регистры РгФ разбиты на две группы (рисунок 3.4). Первая группа регистров (0–223) используется для окон процедур и составляет область стековых регистров, вторая группа (224–255) представляет собой область глобальных регистров. Выделение окон осуществляется по кольцевому принципу. После достижения адреса 223 формируется очередной адрес 0. Вновь выделенное окно может включать несколько последних регистров предыдущего окна, они рассматриваются как выходные регистры окна и служат для передачи фактических параметров запускаемой процедуре.

В каждом окне может быть организована «вращающаяся область», которая предназначена для циклической обработки элементов массива, записываемых из буфера предварительной подкачки в РгФ. При ее организации задаются база относительно окна, текущая база и размер.

Рисунок 3.4 – Логическая структура регистрового файла

Вращающаяся область представляет собой последовательность нескольких смежных стековых регистров, обращение к которым происходит по кольцевому принципу: после записи в последний регистр области очередной элемент массива будет записан в первый регистр. Вращающаяся область может быть организована и в глобальных регистрах. Под нее отводятся последние восемь регистров.

Основной особенностью регистрового файла является возможность обслуживания в течение такта всех его абонентов. Его объем позволяет обеспечить необходимым количеством регистров несколько процедур.

4 Реализация сборки мусора для новых платформ

4.1 Принцип платформенных реализаций

В процессе сборки мусора, реализованным Хансом Боэмом, предусмотрены архитектурно-зависимые части, в которых описывается необходимая информация для получения доступа к памяти для различных операционных систем и архитектур процессоров.

Данные архитектурно-зависимые файлы не влияют работу основного алгоритма по очистке памяти, таким образом они служат для реализации доступа к стекам памяти и регистрам процессора.

По основному назначению сборщик мусора предназначен для сканирования стеков, что невозможно реализовать компилятором языка С. Сложность портирования процесса сборки мусора под новую архитектуру варьируется в зависимости от необходимого функционала сборщика. Но помимо этого важно учитывать особенности архитектуры, если они не учтены в других уже портированных аналогах.

На основе поставленной задачи и изученного материала по устройству процессора можно сделать вывод, что сложность задачи повышается за счет особенностей архитектуры, а также за счет необходимости обращения к памяти, используя язык ассемблера архитектуры «Эльбрус».

В портируемой реализации сборки мусора активно используются #ifdef директивы, по средствам которых осуществляется исполнение участков кода, в зависимости от определенной операционной системы и архитектуры процессора.

Помимо реализации доступа к памяти системы необходимо определить функционал сборщика мусора, а именно:

- *Поддержка потоков*. Данная опция служит для осуществления многопоточного программирования, её поддержка сигнализирует сборщику мусора о необходимости приостановки работы всех потоков для осуществления

процесса работы с памятью. Реализация поддержки многопоточности может быть различна. Данная функциональность напрямую зависит от выбранной системы.

- *Поддержка динамических библиотек.* Помимо правильного определения указателей на участки памяти необходимо указать возможно ли использование динамических библиотек программой. Если программа использует только статические библиотеки, а области памяти не содержат переменных и стеков, определенных в динамических библиотеках, то поддержка данного функционала необязательна.

- *Инкрементальная поддержка.* Данный функционал служит для определения границ обрабатываемой памяти путем вычета минимального размера страницы из начального положения до момента выхода за стек.

4.2 Структура архитектурно-зависимых частей

Архитектурно-зависимые файлы можно разделить на три основные группы:

- *конфигурационные файлы* служат для определения основных макросов системы;

- *файлы процесса сборки мусора*, в которых при помощи директив выполняются участки кода для данной архитектуры;

- *файлы архитектуры системы* содержат наборы функций, реализованных под конкретную архитектуру. Компиляция данных файлов осуществляется только после определения архитектуры системы на этапе сборки программы.

На рисунке 4.1 представлена общая структура архитектурно-зависимых файлов сборщика мусора.

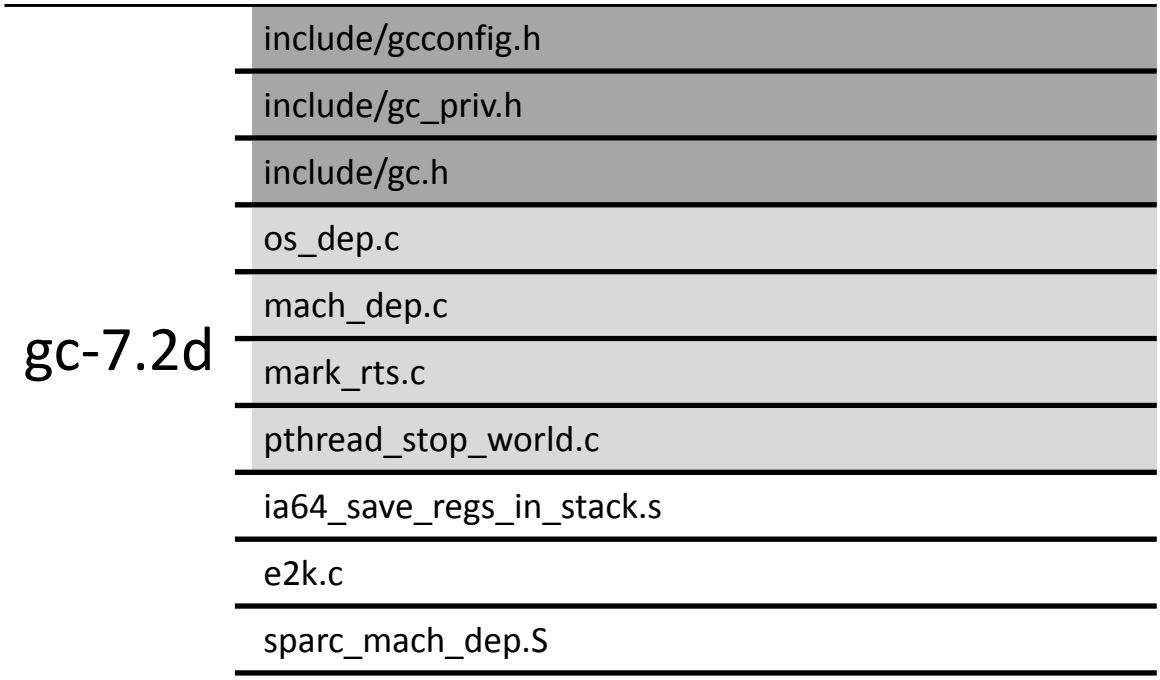

Рисунок 4.1 – Структура архитектурно-зависимых частей сборщика мусора

К первой группе конфигурационных файлов можно отнести заголовочные файлы, в которых необходимо определить основные макросы операционной системы и архитектуры процессора (include/gcconfig.h, include/gc.h, include/gc_priv.h).

Определенные раннее макросы используются уже в программном коде (os_dep.c, mach_dep.c, mark_rts.c, pthread_stop_world.c), а именно в файлах процесса сборки мусора.

Но для некоторых архитектур недостаточно программной реализации в основных файлах сборщика мусора, поэтому создаются файлы для конкретной архитектуры, в которых содержится описание получения доступа к памяти путем программной реализации на языке Ассемблера. Например, для архитектуры микропроцессора IA64 реализован файл ia64_save_regs_in_stack.s, для SPARC – sparc_mach_dep.S. Для решения поставленной задачи о реализации доступа к процедурному стеку под архитектуру микропроцессора «Эльбрус» создан файл e2k.c.

4.3 Описание операционной системы и процессора в заголовочных файлах (gcconfig.h, gc_priv.h, gc.h)

Первоначально при портировании сборщика мусора под новую архитектуру необходимо определить основные макросы под выбранную операционную систему и процессор.

Конфигурационный файл *gcconfig.h* состоит из трех основных частей:

- Раздел, который определяет внутренние макросы сборщика мусора, которые идентифицируют архитектуру (например, IA64 или I386) и операционную систему (например, LINUX или MSWIN32). Обычно это совершается путем тестирования предопределенных макросов. Определяя свои собственные макросы вместо того, чтобы использовать определенные автоматически на стадии сборки, можно избежать проблем при компиляции и дальнейших ошибок.

- Раздел, который определяет основные макросы для конкретной платформы, которые в дальнейшем используются непосредственно в процессе сборки мусора. Описание данных макросов приводится далее. Описываемая часть конфигурационного файла содержит подраздел для каждой архитектуры. В каждом подразделе, как правило, содержатся некоторые архитектурно-зависимые поля, а затем определяются возможные операционные системы, используемые для данной архитектуры процессора.

- Раздел, который заполняет значения по умолчанию для некоторых макрокоманд, которые остались неопределенными в предыдущем разделе, и определяет некоторые другие макросы, которые не так часто используются и редко нуждаются в корректировке для новых платформ. Если осуществляется портирование операционной системы, которая ранее была не реализована, вполне вероятно, что будет необходимо добавить еще один пункт к определению – GET_MEM.

Следующие макросы должны быть грамотно определены для каждой архитектуры и операционной системы:

MACH_TYPE – определяет архитектуру процессора системы. Обычно для идентификации макроса используется только имя архитектуры.

OS_TYPE – определяет имя операционной системы.

CPP_WORDSZ – макрос, определяющий размер машинного слова в битах; используется как постоянный параметр для дальнейших препроцессорных испытаний. В настоящее время определяется в основном либо как 64, либо как 32. В зависимости от разрядности системы может определяться автоматически.

ALIGNMENT – определяет выравнивание данных в оперативной памяти компьютера. Для всех современных 32-разрядных платформ, этот параметр равен 4. Для всех современных 64-разрядных платформ, равен 8.

DATASTART – начало основного сегмента данных. Сборщик мусора будет отслеживать всю память между DATASTART и DATAEND для корневых указателей. На некоторых платформах данный макрос может быть определен как постоянный адрес.

SEARCH_FOR_DATA_START – если макрос DATASTART будет определен как параметр, вычисляющийся динамически, но при этом не был успешно определен, то будет использоваться данный макрос. Такой принцип часто работает на Posix-подобных платформах.

DATAEND – конец основного сегмента данных. По умолчанию определяется как конец основного стека данных.

DATASTART2, DATAEND2 – для некоторых платформ реализовано несколько основных сегментов данных, в таком случае необходимо определить данные макросы.

STACK_GROWS_UP – данный макрос определяется в том случае, если стек растет в сторону больших адресов.

STACKBOTTOM – определяется наибольшим адресом в стеке.

LINUX_STACKBOTTOM – может быть определен вместо STACKBOTTOM, в таком случае информация о дне стека будет определяться из каталога */proc*.

HEURISTIC1, HEURISTIC2 – служат также для альтернативного определения STACKBOTTOM.

DYNAMIC_LOADING – определение данного макроса служит для поддержки динамических библиотек.

PREFETCH, PREFETCH_FOR_WRITE – используется сборщиком мусора для предварительной загрузки в кэш.

HEAP_START – используется в качестве подсказки для дальнейшего определения рабочей области памяти.

ALIGN_DOUBLE – определяется для архитектур, использующих выравнивание по двойному машинному слову.

Для портируемой архитектуры E2K микропроцессора «Эльбрус» и операционной системы Linux были определены следующие макросы:

- определение имени архитектуры системы MACH_TYPE E2K;
- определение имени операционной системы OS_TYPE LINUX;
- выравнивание данных в оперативной памяти ALIGMENT 8;
- размер машинного слова в битах CPP_WORDSZ 64;
- определение дна стека из каталога /proc LINUX_STACKBOTTOM;
- поддержка динамических библиотек DYNAMIC_LOADING;
- определение регистрового стека BACKING_STORE_BASE GC_register_stackbottom;

На переопределении основных макросов сборщика мусора платформенная реализация не заканчивается. В силу того, что для архитектуры микропроцессора «Эльбрус» необходимо реализовать доступ к регистровому файлу, который расположен в процедурном стеке, нужно указать ряд дополнительных параметров, свойственных данной архитектуре.

Для архитектур IA64 и SPARC была реализована функция получения доступа к регистровому стеку *ptr_t GC_save_regs_in_stack(void)*. Описание

данной функции содержится в файле архитектуры системы, но для корректного использования необходимо также дать описание в заголовочных файлах (gc_priv.h, gc.h).

Вызов данной функции передает необходимую информацию сборщику мусора для совершения маркировки объектов.

В силу того, что в архитектуре «Эльбрус» предусмотрен стек, содержащий аппаратурную информацию, а именно регистровый файл, необходимо также передать сборщику мусора все необходимые данные об этом сегменте памяти. Для этого предусмотрена функция *GC_EXTERN ptr_t GC_save_regs_ret_val*, которая возвращает указатель на начало вспомогательной памяти (файл *gc_priv.h*).

В силу того, что для архитектуры «Эльбрус» необходимо реализовать нестандартный доступ к памяти, необходимо переопределить основную структуру данных, предназначенную для описания стека данных. Структура *GC_stack_base* описывается в заголовочном файле *gc.h*.

4.4 Реализация доступа к процедурному стеку для архитектуры «Эльбрус»

В силу особенностей архитектуры микропроцессора «Эльбрус» для передачи полной информации о обрабатываемых участках памяти нужно грамотно определить границы процедурного стека и осуществить его передачу алгоритму сборки мусора. Для реализации доступа к процедурному стеку следует изучить операции, реализованные для выполнения *процедурных переходов*.

Процедурный переход заключается в запуске новой процедуры из некоторой точки текущей процедуры, который может произойти по сигналам прерывания или по командам программы. В первом случае осуществляется вход в аппаратурный обработчик, программные операции позволяют осуществить переход к любой процедуре.

Процедурный переход в общем случае включает следующие действия:

- Вызов в буфер команд и устройство управления строки программного кода новой процедуры;

- Запоминание всей информации о текущей процедуре, необходимой для восстановления состояния процессора на момент вызова процедуры;

- Организацию в РгФ окна для запускаемой процедуры, включающего регистры, содержащие фактические параметры (для процедуры с параметрами);

- Переключение окна стека пользователя; для вызванной процедуры в памяти организуется новый стек пользователя, удаляемый при возврате из процедуры;

- Переключение контекста процедуры.

Возврат из процедуры также является разновидностью процедурного перехода. Все необходимые данные для возврата к продолжению прерванной процедуры считываются из стека связующей информации.

Процедурный переход по командам программы делится на два этапа: подготовка перехода и фактический переход. Подготовка перехода рассмотрена ранее, она реализуется одной из нескольких операций в зависимости от способа получения информации о запускаемой процедуре. Фактический процедурный переход выполняется операцией вызова CALL.

Операция вызова процедуры (CALL). Вызов задается в слоге CS1 кодом операции и кодом wbs — расстоянием в квадрословах от базы текущей процедуры до базы запускаемой процедуры, которое в дальнейшем обозначается символом Δ.

При выполнении операции в регистровой части стека связующей информации организуется новое окно путем выделения очередных двух регистров CR. Одновременно модифицируется указатель аппаратурной вершины стека связующей информации УВСС (PCSHTP), содержащий количество занятых регистров в файле CF. Запоминается информация о текущей процедуре: указатель команды продолжения процедуры, ее контекст (индекс дескрипторов модуля компиляции (CUIR), в который входит текущая

процедура), база окна в РгФ (код wbs), флаг наличия в ее области вещественных 80, состояние предикатного файла, дескриптор стека пользователя, база вращающейся области и регистр состояния процессора. Вся эта информация упаковывается в два квадрослова.

В случае исчерпания файла связующей информации инициируется процесс откачки, после чего выполняются указанные действия. После запоминания информации для возврата происходят переключение указателя команды на адрес начала новой процедуры, формирование дескрипторов окна связующей информации, переключение модуля компиляции (обновление при необходимости спецификаторов) и другие действия.

База окна в РгФ для запускаемой процедуры определяется прибавлением значения Δ к базе текущего окна. Суммирование осуществляется по модулю, равному количеству отводимых на АВС пар регистров (квадрослов). С выделением окна происходит приращение указателя УВС (PSHTP) на величину Δ, чем фиксируется положение дна регистровой части стека процедур.

Операция возврата из процедуры (RETURN). Операция выполняет подготовленную передачу управления запустившей процедуре. Порядок возврата подчиняется стековой дисциплине, то есть переход осуществляется на последнюю неоконченную процедуру.

Действия, которые выполняются в ходе операции возврата, идентичны рассмотренным ранее, но все данные считываются из регистров файла связующей информации. Так, на регистр указателя команды записывается адрес команды, с которой необходимо продолжить выполнение прерванной процедуры, регистры состояния процессора также восстанавливаются данными, находящимися в регистрах CR, регистры группы модуля компиляции, если номер модуля меняется, считываются из таблицы модулей компиляции.

База окна в РгФ процедуры, в которую выполняется переход (назовем ее для краткости процедурой возврата), определяется вычитанием значения Δ из текущей базы, при этом на такую же величину уменьшается УВС (PSHTP.ind).

Переключение фрейма обеспечивает восстановление дескриптора свободной области стека пользователя на момент, предшествующий запуску процедуры, из которой осуществляется возврат. При выполнении возврата восстанавливаются предикатный файл, регистр состояния процессора и флаги, определяющие режим работы прерванной процедуры.

Аппаратурные операции формируются устройством управления и выдаются в основной конвейер на стадии базирования адресов операндов. Рассмотрение аппаратурных операций в этом месте обусловлено методическими соображениями — они отсутствуют в программном коде и формируются при возникновении соответствующей ситуации.

Системой команд предусмотрены три такие операции. Одна из них имитирует вставку в широкую команду «пустых» простых команд (операция BUBBLE), две другие осуществляют откачку (SPILL) данных из аппаратурной вершины стека (стека процедур или стека связующей информации) и подкачку (FILL) в нее данных из части стека, находящейся в памяти.

Операция откачки. Инициируется операцией очистки стека процедуры, стека связующей информации или возникает при выполнении операции ввода в новую процедуру в случае нехватки ресурса свободных регистров при организации окна файла (RF или CF).

Поскольку структуры стеков процедур и связующей информации одинаковы, принцип выполнения операции откачки рассмотрим применительно к стеку процедур. Применение ее основано на циклическом выделении ресурса регистрового файла при образовании окон: после использования регистра с номером 0x6f следующим номером будет 0x00 (границы окон выровнены по квадрослову, поэтому количество пар регистров равно 112).

Для откачки используются указатели аппаратурной вершины стека УВС (PSHTP) и УС (PSR). Первый из них указывает на дно стека процедур — регистр, начиная с которого происходит откачка. В указателе стека имеется поле, где хранится индекс начала свободной области. Откачка производится с

точностью до двух регистров РгФ. Детали процесса откачки связаны с особенностями хранения данных в АВС и памяти.

Для проведения передачи данных о процедурном стеке был реализован программный код, осуществляющий операцию откачки применимую для процедурного стека. Данная реализация описана в функции *ptr_t GC_save_regs_in_stack(void)*, которая возвращает указать на вершину процедурного стека УВС.

Для осуществления работы с процедурным выполнено копирование всех регистровых окон из процедурного стека путем реализации функции *size_t GC_e2k_get_procedure_stack (__u64 **ps_buf)*. Данная функция осуществляется копирование всех регистровых окон кроме текущего в указанный на входе буфер и возвращает указатель на вершину стека.

4.5 Определение границ обрабатываемой памяти (os_dep.c)

После реализации доступа ко всем участкам памяти во вспомогательном файле для архитектуры «Эльбрус» *e2k.c* необходимо осуществить грамотную передачу данных процессу сборки мусора.

Как было описано ранее в сборщике мусора предусмотрены архитектурно-зависимые участки кода, реализованные для конкретной архитектуры.

Для нахождения дна стека используется функция *GC_find_limit(GC_save_regs_in_stack(), FALSE)*, которая определяет указатель на самый старший адрес стека путем прибавления минимального размера страницы к указателю текущей вершины до момента выхода за границу стека. Сигналом перехода за границу стека служит появление ошибки сегментации, которую отслеживает внутренний обработчик сборщика мусора.

Как было описано ранее, границы основного стека данных можно определить различными способами в зависимости от особенностей архитектуры системы. В случае архитектуры микропроцессора «Эльбрус»

данную информацию можно получить из каталога /proc, для этого необходимо также указать использование функции *static ptr_t backing_store_base_from_proc(void),* которая осуществляет получение данных о границах основного стека из каталога */proc/self/maps.*

4.6 Передача данных о границах процедурного стека (mach_dep.c)

Перед началом выполнения процесса автоматической сборки мусора коллектору необходимо передать информацию о всех участках памяти. Для архитектур таких как IA64, SPARC и E2K нужно указать границы данные о стеках, содержащих регистры. В терминологии сборщика мусора данный сегмент памяти называется регистровым стеком.

До момента выполнения алгоритма сборки мусора количество уже выполненных процессов минимально, поэтому на предварительной стадии осуществляется вызов функции *ptr_t GC_save_regs_in_stack(void)*, которая возвращает указать на вершину процедурного стека УВС. Таким образом, получена нижняя граница регистрового стека.

4.7 Операция «Остановки мира» (pthread_stop_world.c)

Процесс сборки мусора, реализованный Хансом Боэмом, осуществляет запуск алгоритма в момент остановки работы всех потоков. Данная реализация процесса описывается принципом «остановки мира».

Таким образом, вызов функции *size_t GC_e2k_get_procedure_stack (__u64 **ps_buf)*, которая осуществляет копирование всех регистровых окон кроме текущего в указанный на входе буфер и возвращает указатель на вершину стека, осуществляется на данном этапе алгоритма.

Для совершения процесса управления памятью сборщику мусора передается информация о границах процедурного стека, его размер и рабочая область регистрового файла, полученная путем копирования в буфер.

4.8 Маркировка участков памяти (mark_rts.c)

В момент выполнения фазы маркировки сборщику мусора передается информация о всех сегментах обрабатываемой памяти. Таким образом, происходит передача информации о стеке вызовов, управляемой кучи и регистров процессора.

5 Тестирование реализации и анализ результатов

5.1 Проведение внутренних тестов сборщика мусора

В портируемом сборщике мусора предусмотрены внутренние тесты, на основе которых можно сделать вывод о правильности переноса процесса под выбранную операционную систему и архитектуру микропроцессора.

Внутренняя система тестирования сборщика мусора включает в себе два основных теста:

- *setjmp_test* определяет архитектуру процессора и операционную систему компьютера, а также выводит данные об адресах обрабатываемых сегментов памяти, о размере машинного слова и о выравнивании данных в оперативной памяти компьютера;

- *gctest* осуществляет выполнение программного кода, который проверяет весь функционал сборщика мусора для грамотного освобождения памяти. Результатом работы теста является определение числа распознанных объектов среди которых выявляются некорректные, атомарные и подлежащие дальнейшему удалению. По окончанию работы теста выводятся итоговые значения о числе удаленных объектов и финальном размере обрабатываемой области памяти.

Проведение *setjmp_test* показало следующие результаты:

- MACH_TYPE E2K – архитектура процессора «Эльбрус»;

- OS_TYPE LINUX – операционная система Linux;

- STACK_GROWS_UP – стек вызовов растет вверх;

- STACKBOTTOM 0xc2dfff881000 – определен наибольший адрес стека;

- ALIGMENT 8 – выравнивание данных в оперативной памяти по 8 байту;

- WORDSZ 64 – размер машинного слова равен 64 битам.

Исходя из полученных результатов можно сделать вывод, что сборщик мусора определил значение всех данных, параметров и адресов системы верно.

5.2 Сравнение результатов на Эльбрусе и на Intel

В силу того, что проведение *gctest* представляет собой исполнение программного кода, в момент исполнения которого происходит процесс сборки мусора, результаты тестирования необходимо сравнить с итогами на других архитектурах.

В качестве эталонного решения выбрана реализация сборки мусора для архитектуры x86 процессора Intel. Выбор обусловлен тем, что разработчик сборщика мусора утверждает о полном портировании процесса под данную архитектуру.

Результаты проведенных тестирований отражены в таблице 1, где E2K – архитектура микропроцессора «Эльбрус», а x86 – архитектура микропроцессора Intel.

Таблица 1 – Результаты проведения внутренних тестов

	Помеченные объекты	Непомечен-ные объекты	Атомарные объекты	Удаленные объекты	Финальный размер кучи
E2K	748193	202	1250000	21760	11612160
x86	748193	202	1250000	21760	17887232

5.3 Выводы по проведенным тестированиям

Так как внутренние тесты представляет собой запуск процесса сборки мусора во время исполнения конкретного программного кода, отображенные результаты в таблице 1 для архитектуры Эльбрус и x86 оказались идентичными, что говорит об успешном портировании сборщика мусора под архитектуру "Эльбрус".

Единственный параметр, который оказался различным, это размер управляемой кучи. Такой результат справедлив в силу того, что размер адресного пространства выделяемый под процесс варьируется самим процессором.

6 Кроссплатформенная сборка

Кроссплатформенная сборка предназначена для создания универсальных сценариев сборки программных пакетов, адаптируемых для различных архитектур.

Для разработчиков дистрибутива для микропроцессоров «Эльбрус» предусмотрена собственная система кросс-сборки. В ней осуществляется сборка пакетов для различных архитектур, проведение тестирований и дальнейшая отладка и оптимизация программного обеспечения.

6.1 Особенности кроссплатформенной сборки для архитектуры «Эльбрус»

Система кросс-сборки включает в себе:

- сервер, на котором происходит непосредственная работа;

- движок сборочной системы;

- рабочее пространство, содержащее функциональные скрипты и конфигурацию;

- репозиторий – общедоступное хранилище с поддержкой контроля версий;

- исходники программных пакетов, которые необходимо портировать под архитектуры микропроцессоров «Эльбрус»;

- сценарии сборки, на основе которых происходит создание готовых *.deb* пакетов.

Движок сборочной системы написан на языке программирования Python и использует модульную структуру команд. Для разработчиков важны исполняемые файлы движка и внутренние модули-команды.

Принципы построения команд:

- Каждая команда представляет собой модуль, написанный на интерпретируемых языках (Python, bash);

- Для каждого модуля создается директория с его названием;

- Подмодули располагаются в директориях их родительских модулей;

- Каждый подмодуль имеет свою директорию, если имеет дочерние подмодули;

- Команды могут иметь свои входные параметры, которые обычно передаются через запятые;

Пример команды установки движка:

rcr settle engine dir_name,

где rcr – исполняемый файл кросс-системы,

settle – родительская команда,

engine – команда установки движка,

dir_name – путь к директории, в которую будет происходить установка.

По описанному принципу осуществляется работа разработчиками программного обеспечения в кросс-системе сборки.

6.2 Уровни оптимизации компилятора

В системе сборки программного обеспечения предусмотрен собственный компилятор, реализованный для архитектур микропроцессора «Эльбрус».

Компилятор - это интерфейс, используемый при запуске из командной строки, и программа, сформированная из исходного текста.

При запуске задаются файлы, составляющие исходный текст и его тип, то есть транслируемый язык, результирующие файлы, представляющие объектный код и его вид, задающий машину, для которой создается программа. Кроме того, гибкий механизм управления трансляцией, обрабатывающий опции и скрипт-файл, определяет особенности данного запуска компилятора. Работу программы компилятора можно разбить на несколько основных фаз:

- *Фронтальная часть* (англ. frontend) осуществляет синтаксический и семантический разбор исходной программы с диагностикой ошибок и строит

атрибутное синтаксическое дерево исходной программы с контекстными таблицами, что является основой для последующих фаз компилятора.

- *Фаза анализа* строит информационные структуры и графы (информационно-управляющие и вызовов). Эти структуры используются последующими фазами для моделезависимых и моделенезависимых оптимизаций исходной программы.

Выполняется глобальный контекстный межпроцедурный анализ для обнаружения потенциальной информационной зависимости для динамически вычисляемых адресов и выявления итерационной зависимости циклических участков.

- *Фаза моделенезависимых оптимизаций* выполняет удаление избыточных вычислений, сборку общих подвыражений, свертку константных выражений, упрощение операций (покадровую оптимизацию), обнаружение индуктивных переменных и инвариантов циклов, а также собирается дополнительная информация о спецификации данных с помощью потокового анализа.

- *Фаза моделезависимых оптимизаций* использует специфические особенности целевой платформы для получения оптимального объектного кода, включая параллельность на уровне операций, предикатный и спекулятивный режим исполнения операций. При оптимизации циклических участков используется аппаратная поддержка техники совмещения итераций циклов.

- *Фаза компактирования* "проецирует" построенное и преобразованное на предыдущих фазах промежуточное представление на структуру широкого командного слова, а также распределяет необходимые ресурсы, включая регистры, буферные памяти, шины данных, каналы доступа в память и т.п.

- *Фаза создания файла объектного кода* (ФОК) завершает оформление кода в стандартизированный файл. При этом в файл включаются некоторые информационные структуры, нужные для технологических этапов разработки программы: символьная отладка, аварийная выдача, комплексация и т.п.

Используемый компилятор имеет различные уровни оптимизации. *Оптимизация программного кода* — это модификация программ, выполняемая оптимизирующим компилятором или интерпретатором с целью улучшения их характеристик, таких как производительности или компактности, — без изменения функциональности.

При подаче опций при сборке пакетов можно выделить четыре уровня оптимизации:

- O0 – компилятор выполняет прямую компиляцию, не изменяя порядка следования инструкций, и не предпринимая никаких других попыток оптимизации;

- O1, O2 – оптимизация должна выполняться практически без изменения порядка следования инструкций. Применяются только явно разрешенные (например, директивой #pragma) средства оптимизации.

- O3 – данный флаг разрешает компилятору применять все доступные средства оптимизации и снимает ограничения на объем памяти, выделенной компилятору.

Сборка garbage collector-а осуществлялась с уровнем оптимизации O3.

6.3 Сборка под различные архитектуры (E2K, x86, SPARC)

Для осуществления сборки программного пакета под различные архитектуры необходимо разработать сценарий, по которому будет проходить создание готового установочного пакета.

В системе кросс-сборки микропроцессоров «Эльбрус» предусмотрен основной шаблон создания сценариев сборки. Таким образом, любой сценарий должен состоять из трех обязательных секций, соответствующим этапам сборки пакета:

- *src_config* – функция, описывающая правила конфигурирования пакета, содержащая вызов интерфейса оболочки econf;

- *src_compile* – функция, описывающая правила сборки пакета, содержащая вызов интерфейса-оболочки emake;

- *src_install* – функция, описывающая правила сборки пакета, содержащая вызов интерфейса-оболочки install;

Все сценарии сборки пакетов реализовываются на скриптовом языке программирования bash.

По завершении работы в системе кросс-сборки был разработан универсальный сценарий для создания установочного пакета сборщика мусора gc-7.2d для архитектур E2K (“Эльбрус”), SPARC и x86.

7 Использование и дальнейшая оптимизация решения

Реализованный алгоритм сборки мусора под архитектуру «Эльбрус» используется не только программистами при разработке программ, но и сторонними приложениями.

Boehm Garbage Collector входит в состав компилятора GNU Compiler for Java, а также в кросс-платформенное программное обеспечение Portable.NET. Процесс автоматической сборки мусора для языков C и C++ также используют и другие языки программирования, например, такие как: Guile и Embeddable Common Lips.

Язык программирования Guile входит в состав дистрибутива микропроцессоров семейства «Эльбрус», поэтому на примере использования данного программного пакета можно проверить работоспособность портируемого сборщика мусора.

7.1 Применение сборщика мусора на сторонних приложениях

В качестве анализа работы сборщика мусора выбран программный пакет guile-2.0.13. Данное приложение является реализацией языка программирования Scheme – Guile; рекомендован в качестве встраиваемого языка в программные продукты проекта GNU.

Выбор продукта Guile обусловлен тем, что данный программный пакет входит в зависимости утилиты, которая служит для создания конфигурационных скриптов – Autoconf.

Autoconf автоматически настраивает пакеты с исходным кодом для работы в Unix-подобных операционных системах. Данная утилита входит в состав дистрибутива микропроцессоров «Эльбрус», так как основной операционной системой семейства процессоров является LINUX.

Язык программирования Guile в качестве автоматического процесса сборки мусора использует реализованный для архитектуры микропроцессора «Эльбрус» boehm garbage collector.

Для проведения анализа работы программного пакета guile-2.0.13 использовался профилировщик perf. Утилита perf служит для осуществления анализа производительности отдельных подсистем исследуемой программы. При помощи профилировщика можно отследить конкретные функции, библиотеки и участки программы, на которые приходятся максимальные временные задержки. В качестве результата работы утилиты perf предоставляется отчет о работе программы. Используя опции профилировщика можно отслеживать интересующие характеристики и параметры работы исследуемой программы.

При анализе работы программного пакета guile основной интерес вызывало время, отведенное на работу портируемого сборщика мусора.

Отчеты профилирования работы guile показали результаты, отраженные в таблице 2.

Таблица 2 – Результаты профилирования работы guile до оптимизации

Расходы	Программа	Библиотека	Главная функция
24,40 %	guile	libgc.so.1.0.3	GC_mark_from
19,34 %	guile	libguile-2.0.so.2	vm_debug_engine
4,18 %	guile	libguile-2.0.so.2	vm_regular_engine
2,35 %	guile	libguile-2.0.so.2	scm_hasher
2,30 %	guile	libgc.so.1.0.3	GC_header_cache_miss

Из анализа результатов, приведенных в таблице 2, можно сделать вывод, что наибольшие временные расходы работы приложения guile отводятся на выполнение функции маркировки GC_mark_from сборщика мусора.

Исходя из профилирования работы сторонней программного пакета guile можно сделать вывод о необходимой оптимизации портируемого процесса сборки мусора boehm garbage collector.

7.2 Оптимизация решения на основе результатов эксплуатации

После проведения исследований по работе портируемого сборщика мусора была поставлена задача об оптимизации решения для осуществления ускорения работы процесса сборки мусора на этапе маркировки.

Как описано ранее, на фазе маркировки помечаются все возможно достижимые объекты. Данный процесс является затратным по временным характеристикам в силу того, что сборщик мусора не имеет информации о местонахождении указателей переменных в массиве данных, поэтому производится рассмотрение всех статических областей данных, стеков и регистров как потенциально содержащих указатели. Любые комбинации битов, которые представляют адреса внутри массива данных объектов, рассматриваются сборщиком мусора как указатели.

В архитектуре «Эльбрус» предусмотрена защита контекста программного модуля, образованный объектами, которые допускается в нем использовать согласно правилам языка программирования. Это осуществляется путем организации доступа к низкоуровневому представлению объекта в памяти или регистрах через «проходную» его дескриптора — служебного слова, содержащего ссылку и информацию об объекте, соответствующую его типу. Доступ возможен только через дескриптор. Аппаратные операции создания и использования дескрипторов специализированы таким образом, что не оставляют возможности непосредственного воздействия на объект.

Принципиальным фактором защиты, реализуемым через аппаратуру, является тегирование. Оно выполняется путем добавления к информационным разрядам дополнительного поля — тега, определяющего тип данных. Каждое 4-байтовое слово в памяти, регистрах и шинах сопровождается 2-разрядным тегом.

Тег слова кодирует следующие признаки:

0 — слово содержит числовую информацию либо само по себе, либо являясь фрагментом формата;

1 — слово содержит нечисловую информацию с форматом одинарного слова;

2 — слово содержит фрагмент нечисловой информации формата двойное слово;

3 — слово содержит фрагмент нечисловой информации формата квадро.

Ни длина, ни тип числовых данных архитектурно не различимы. Семантическое наполнение числовой переменной отслеживается компилятором и проявляется, когда она становится операндом какой-либо операции.

В отличие от числовых данных, нечисловые данные строго типизируются. Прежде всего гарантируется целостность составных форматов (двойное слово и квадро), что предполагает следующее:

- все фрагменты должны иметь тип, соответствующий формату;

- при любых манипуляциях фрагменты должны сохранять свой порядок.

Второе качество реализуется выровненным расположением данных как в памяти, так и в регистровом файле микропроцессора. Так, младшее слово адресной переменной с форматом двойного слова всегда помещается по адресу, кратному 8 байтам, а старшее — следом за ним, смежным образом. Пересылки отдельных фрагментов приводят к разрушению их типов (превращению в неспециализированные данные). Операции, требующие специализированных операндов (например, адресных данных), строго контролируют их тип и целостность.

В рамках каждой из форматных групп конкретный тип специализированных данных определяется значением выделенной группы информационных разрядов — внутреннего тега.

Кроме обычных дескрипторов, описывающих регулярные массивы в памяти, введены специальные дескрипторы, позволяющие описывать структуры данных в соответствии с требованиями объектно-ориентированного программирования. Это существенно упрощает доступ к таким данным. Дескриптор объекта содержит описание открытых, защищенных и приватных областей данных.

Регистры контекста выполняемой программы дополнены регистрами индекса текущего модуля компиляции, дескриптора текущего модуля компиляции, дескриптора глобальных данных текущего модуля компиляции, дескриптора типов и текущего типа. Введены операции обращения в память с использованием этих дескрипторов. Все дескрипторы имеют внутренние теги, позволяющие контролировать корректность их использования (только в специальных операциях обращения в память и преобразований в сторону сокращения прав их использования).

Реализация поддержки распознавания тегированных значений на этапе маркировки позволит ускорить процесс сборки мусора и таким образом, оптимизировать работу.

Так как слова, содержащие числовые данные, кодируются тегом со значением «0», то рассмотрение только нулевых тегированных значений на этапе маркировки позволяет ускорить процесс сборки мусора за счет сокращения обрабатываемых данных.

7.3 Результаты проведенной оптимизации

После проведенной оптимизации процесса сборки мусора было проведено повторное профилирование работы программного пакета guile, результаты которого отражены в таблице 3.

Таблица 3 – Результаты профилирования работы guile после оптимизации

Расходы	Программа	Библиотека	Главная функция
27,94 %	guile	libguile-2.0.so.2	vm_debug_engine
13,69 %	guile	libgc.so.1.0.3	GC_mark_from
3,96 %	guile	libguile-2.0.so.2	vm_regular_engine
2,47 %	guile	libguile-2.0.so.2	scm_hasher
2,02 %	guile	libguile-2.0.so.2	scm_i_struct_hash

Из полученного отчета профилирования можно сделать вывод, что проведенная оптимизация сократила расходы времени, приходящиеся на маркировку объектов, более чем на 10 %, что является успешным результатом.

ЗАКЛЮЧЕНИЕ

При выполнении работы были решены поставленные задачи и достигнута заявленная цель. Сборщик мусора полностью портирован под архитектуру микропроцессора «Эльбрус» и добавлен в состав дистрибутива.

Для осуществления поставленной цели в ходе работы были решены следующие задачи:

- реализован доступ сборщику мусора к процедурному стеку путем использования аппаратурных операций откачки регистрового файла;

- осуществлен доступ к стеку вызовов и управляемой куче посредством получения данных из файловой системы, содержащей информацию о системных процессах;

- проведены внутренние тестирования сборщика мусора и анализ полученных результатов путем сравнения с реализациями на других архитектурах;

- реализована кроссплатформенная сборка адаптируемого программного пакета под архитектуры E2K, SPARC и x86;

- проведена эксплуатация сборщика мусора на стороннем приложении;

- на основе профилирования работы стороннего программного приложения проведена оптимизация решения;

- выявлено ускорение работы сборщика мусора с новыми модификациями;

- полностью адаптированный сборщик мусора добавлен в состав дистрибутива.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Велихов Е. П. Сергей Алексеевич Лебедев // Информационные технологии и вычислительные системы. — 2002. — № 3. — 31–35 с.
2. Бурцев В. С. Параллелизм вычислительных процессов и развитие архитектуры ЭВМ. — М.: Торус пресс, 2006.
3. Борисов Ю. И. Проектные центры компьютеростроения в программах обеспечения национальной безопасности // Матер. конф. «Перспективы развития высокопроизводительных архитектур. История, современность и будущее отечественного компьютеростроения». — 2008. — № 1. — 8–14 с.
4. Бабаян Б. А. История развития архитектур вычислительных машин // Ершовские лекции по информатике. — Новосибирск: Изд-во Ин-та информатики им. А. П. Ершова СО РАН, 2009.
5. Ким А. К., Перекатов В. И., Сахин Ю. Х. Развитие и реализация архитектуры вычислительных комплексов серии «Эльбрус» для решения задач ракетно-космической обороны // Вопросы радиоэлектроники. Сер. ЭВТ. — 2010. — Вып. 3. — 5–17 с.
6. В.Л. Григорьев. Микропроцессор i486. Архитектура и программирование. /М.: "МИКАП", 1993 г. – 14 с.
7. А.В. Нестеренко. ЭВМ и профессия программиста. /М.: Просвещение, 1990 – 214-216 с.
8. Boehm, H., and D. Chase, "A Proposal for Garbage-Collector-Safe C Compilation", Journal of C Language Translation 4, 2 (Decemeber 1992), pp. 126-141.
9. Boehm, H., "Simple Garbage-Collector-Safety", Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation.
10. Hans-J. Boehm and Alan J. Demers, «Boehm Garbage Collector C/C++,» 2004.

11. Raymond T. Chen, «Everybody thinks about garbage collection the wrong way,» *IEEE Trans. on Antennas and Propagation*, 2010.
12. Джон Л. Хеннеси, Дэвид А. Паттерсон, «Компьютерная архитектура. Количественный подход,» Перевод с английского М. В. Таранчевой под редакцией к.т.н. А. К. Кима, 2016. – 348 с.
13. The SPARC architecture manual: Version 9. — Englewood Cliffs, N.J.: Prentice Hall, 1993.
14. Hennessy J. L., Patterson D. A. Computer Architecture. A Quantitative Approach. — 4 rev. ed. — Elsevier Science, 2006.
15. Волконский В. Ю., Грабежной А. В., Муханов Л. Е., Нейман-заде М. И. Исследования влияния подсистемы памяти на производительность распараллеленных программ // Вопросы радиоэлектроники. Сер. ЭВТ. — 2011. — Вып. 3. — 22–38 с.
16. Галазин А. Б., Ступаченко Е. В., Шлыков С. Л. Программный метод предварительной подкачки кода в архитектурах со статическим планированием // Программирование. — 2008. — № 1. — 67–74 с.
17. Иванов Д. С. Распределение регистров при планировании инструкций для VLIW-архитектур // Программирование. — 2010. — № 6. — 74–80 с.

ПРИЛОЖЕНИЕ А

Графическая часть магистерской диссертации

В графическую часть магистерской диссертации входят:

1) Структурная схема микропроцессора с архитектурой SPARC (графический лист 1);

2) Блок-схема микропроцессора «Эльбрус» (графический лист 2);

3) Блок-схема алгоритма процесса сборки мусора (графический лист 3);

4) Особенности организации памяти архитектуры «Эльбрус» (графический лист 4);

5) Организация стеков в архитектуре «Эльбрус» (графический лист 5);

6) Организация доступа к сегментам памяти (графический лист 6);

7) Результаты тестирований (графические листы 7-9);

8) Описание оптимизации и сборки программы (графический лист 10);