Министерство образования и науки Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования

«Московский физико-технический институт

(государственный университет)»

Факультет радиотехники и кибернетики

Кафедра теоретической и прикладной информатики

ПОДДЕРЖКА ОБЪЕКТНЫХ ХРАНИЛИЩ ДАННЫХ В ACRONIS BACKUP GATEWAY

Выпускная квалификационная работа

(бакалаврская работа)

Направление подготовки: 03.03.01 Прикладные математика и физика

Профиль обучения: Инфокоммуникационные и вычислительные системы и технологии

Выполнил:
студент 311 группы _____ Потапов Артем Валерьевич

Научный руководитель:
к.ф.-м.н. _____ Анисимов Артем Борисович

Москва 2017

Содержание

Введение

Acronis Backup Gateway (ABGW) – часть системы хранения данных (СХД) Acronis Storage 2.x, которая позволяет использовать СХД для хранения бекапов, создаваемых другими продуктами Acronis. Часть клиентов Acronis уже имеет свои собственные СХД и предпочитает использовать их для размещения бекапов. Зачастую такие СХД предоставляют интерфейс, совместимый с Amazon S3, который стал де-факто стандартом для объектных хранилищ данных.

Как следствие, возникает задача поддержки в ABGW объектных хранилищ данных, таких, как Amazon S3 и совместимые (Swisscom, Internet Initiative Japan и др.). Для сокращения времени разработки ABGW доступ к объектным хранилищам хочется реализовать с использованием официальных SDK от разработчиков S3. Эти SDK предоставляются в основном для Java, Python и Go. Ввиду того, что ABGW, как и другие компоненты Acronis Storage, написан на C, использование каждого из этих SDK потребует взаимодействия модулей, написанных на разных языках программирования. Поэтому, помимо соображений производительности и потребления ресурсов, становится важным вопрос о простоте интеграции Object Storage SDK и ABGW. Данная дипломная работа рассматривает вопрос об использовании Amazon S3 SDK для Go в ABGW. Помимо задачи вызова функций на одном языке из функций на другом, перед нами стоит вопрос о том, как обеспечить взаимодействие программных окружений, которые ожидают ABGW и Go. Каждое из окружений предоставляет свои средства для обеспечения параллелизма и неблокирующего ввода-вывода, и необходимо найти способ сшить эти два механизма.

Постановка задачи

Итак, перед нами стоят следующие задачи:

1) ознакомиться с AWS SDK для Go, написать тестовые программы для совершения операций «Upload» и «Download» с сервисом Amazon S3,

2) убедиться в работоспособности FFI (foreign function interface) для Go,

3) реализовать механизм неблокирующего вызова кода на Go из «libpcs_io-корутин», и обеспечить бесконфликтную работу окружений «libpcs_io» и Go (обработка сигналов),

4) понять, какие имеются инструменты для отладки кода, в котором перемежаются функции на C и на Go.

Глава I. Знакомство с AWS SDK для Go

1.1 Общее описание AWS SDK для Go

AWS SDK для Go предоставляет API и некоторые утилиты, которые могут быть использованы разработчиками в процессе разработки приложений на языке Go, предназначенных для взаимодействия с сервисами, предоставляемыми платформой Amazon Web Services. Рассматриваемая SDK является «оболочкой» над API Amazon Web Services, скрывающей в себе низкоуровневые операции, связанные с аутентификацией, повторными запросами и обработкой ошибок. Также AWS SDK для Go реализует некоторые утилиты, позволяющие совершать параллельную загрузку данных, разбивая отсылаемые данные на части. Это может быть полезно в случае, если приходится иметь дело с большими объектами.

1.2 Настройка среды разработки для Go

Первый шаг - установка пакетов языка Go на компьютер[1][2]. Следующий шаг – загрузка и установка AWS SDK для Go[3]. Стандартными механизмами загрузки, сборки и установки пакетов Go являются инструменты «go». Эти инструменты требует от разработчика определенного способа организации программного кода. Весь Go код должен храниться в определенной директории, именуемой рабочим пространством (workspace). Рабочее пространство хранит в себе некоторое количество репозиториев. Каждый такой репозиторий может содержать один или несколько пакетов Go. Каждый пакет в свою очередь содержит один или несколько исходных файлов, находящихся в одной директории. При необходимости импорта каких-либо

пакетов в проект, путь импорта (import path) определяется положением пакета в вышеописанной иерархической структуре, начиная с директории, являющейся корнем рабочего пространства (то есть с workspace).

Разберемся поподробнее, из чего состоит каталог рабочего пространства. В каталоге workspace присутствуют три каталога: «src», «pkg», “bin”. В каталоге «src» хранятся исходные файлы, в каталоге «pkg» хранятся собранные пакеты (после применения инструмента «go build»), а в каталоге «bin» хранятся исполняемые файлы (после применения инструмента «go install»)[4].

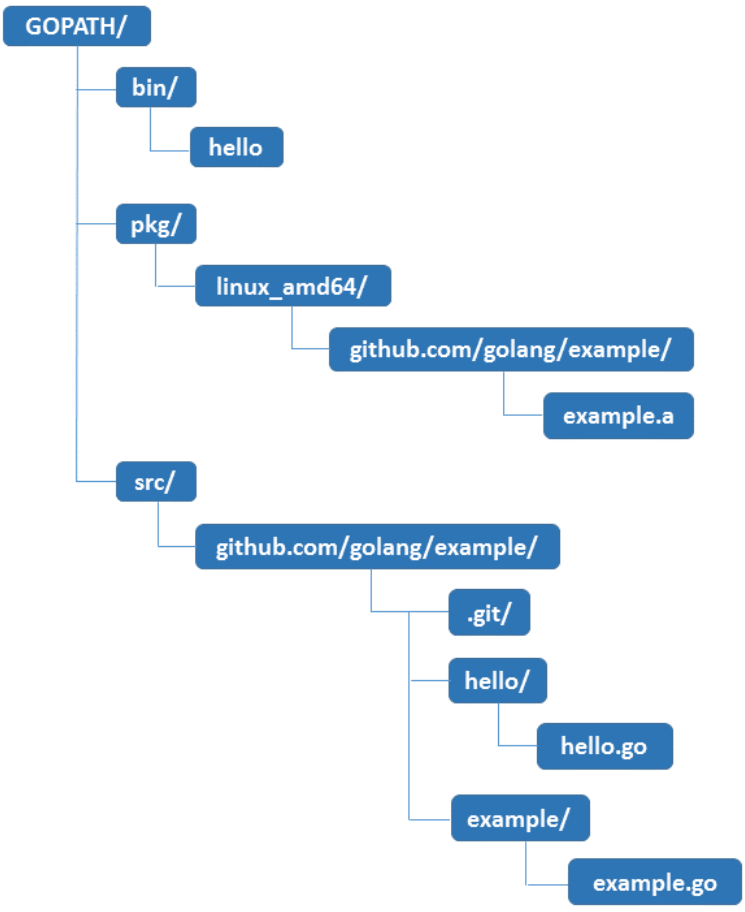

Рисунок 1 – Устройство «рабочего пространства» для разработки на языке Go

Таким образом, для разработки на языке Go в первую очередь необходимо позаботиться о наличии рабочего каталога. Как же Go понимает,

где находится рабочий каталог? Для того, чтобы указать путь до рабочего каталога, используется переменная окружения «GOPATH». По умолчанию данный путь имеет вид: «$HOME/go». Но при желании вполне возможно указать иной путь. Отмечу, что далее в работе я буду указывать имена пакетов рассматриваемой библиотеки относительно директории «$GOPATH/src/github.com/aws/aws-sdk-go/».

1.3 Загрузка и установка AWS SDK для Go

Настроив рабочее пространство вышеуказанным способом, можно переходить к загрузке пакетов AWS SDK для Go. С помощью инструмента «go get» выполняется загрузка исходных файлов и уже собранных пакетов из удаленного репозитория соответственно в каталоги «src» и «pkg». После этой операции пакеты AWS SDK для Go готовы к использованию, но имеется необходимость в совершении некоторых настроек[4].

1.4 Дополнительные настройки AWS SDK для Go

Для использования AWS SDK для Go необходимо иметь идентификатор пользователя (access key ID) и секретный ключ (secret access key)[5]. Этими ключами SDK подписывает запросы, отправляемые в «Amazon Web Services». Также необходимо указывать, в каком регионе будут храниться данные, этот параметр позволит SDK сформировать правильный запрос (в запросе будет указан верный URL). Передачу ключей и идентификатора региона в SDK можно совершить несколькими способами.

Передать библиотеке идентификатор региона можно через переменную окружения (AWS_REGION), либо в коде самой программы при создании

объекта «Gonfig» (структуры, одним из полей которой является строка, отвечающая за регион) из пакета «aws» рассматриваемой библиотеки. Первый способ наиболее удобен, так как библиотека в этом случае самостоятельно выполняет поиск данного параметра. Второй способ удобен в том случае, если необходимо для конкретной программы выполнить индивидуальный запрос в другой регион. В своей работе я использовал первый способ, так как мне была предоставлена всего одна тестовая учетная запись на «Amazon Web Services». В случае, если ни один из перечисленных вариантов передачи идентификатора не использован, библиотека вернет ошибку.

Передачу идентификационных ключей можно выполнить тремя способами: с помощью общедоступного файла с идентификационными ключами, с помощью переменных окружения (AWS_ACCESS_KEY_ID и AWS_SECRET_ACCESS_KEY) и путем передачи данных ключей в коде программы. Последний способ аналогичен передаче идентификатора региона в объект «Config» пакета «aws». Если рассматриваемые параметры не были переданы напрямую в коде программы, то библиотека начинает поиск этих параметров по следующему алгоритму. Сначала производится поиск общедоступного файла с идентификационными ключами, затем, если файл не найдет, производится загрузка данных параметров из переменных окружения. Если оба способа безуспешны, то библиотека возвращает ошибку. Использование общедоступного файла с идентификационными ключами предполагает расположение этого файла в определенном месте файловой системы – в домашнем каталоге. Требуется, чтобы данный файл располагался в скрытой директории «$HOME./aws» и имел имя «credentials». В файле должны присутствовать идентификационные ключи в определенном формате, проиллюстрированном на рисунке 2.

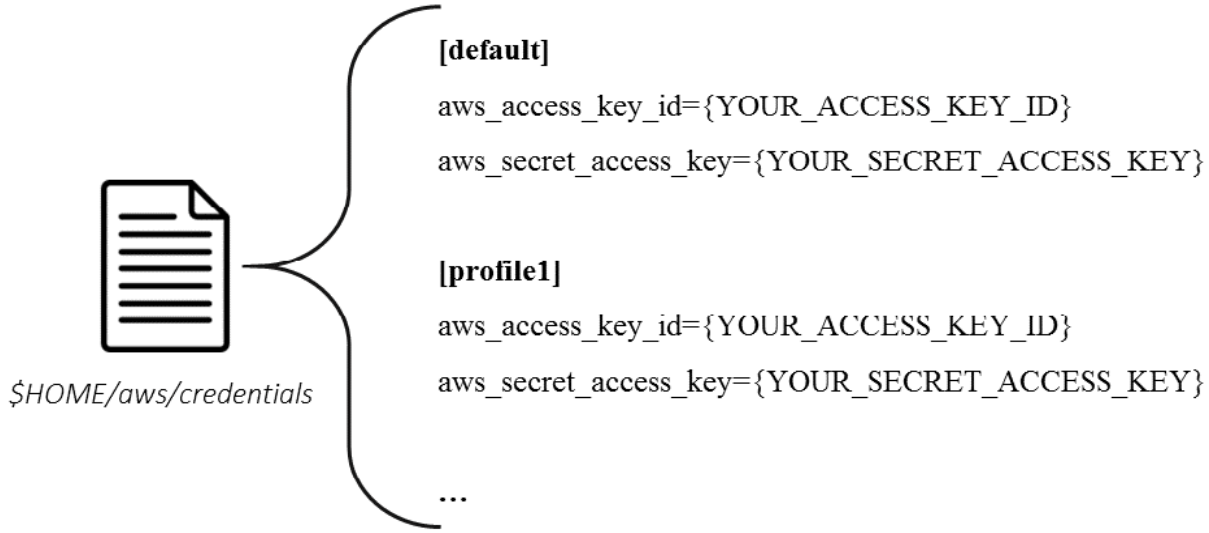

Рисунок 2 – Содержание файла «credentials»

Данный файл также позволяет располагать в нем идентификационные ключи для различных учетных записей AWS, которые могу иметь разное назначение. На приведенном рисунке видно, что первая учетная запись является «учетной записью по умолчанию» и обозначается идентификатором «[default]». Допустим, имеется еще одна учетная запись, являющаяся тестовой. Тогда ее можно добавить в файл, а на месте идентификатора написать, например, «[test]». В таком случае при создании объекта «Config» из пакета «aws» в него следует передать идентификатор учетной записи в виде строки «test». В данной работе я использовал только одну учетную запись, а идентификационные ключи я передаю с помощью файла «credentials». Выполнив все необходимые настройки библиотеки AWS SDK для Go, можно переходить к следующему шагу - изучению пакета «service/s3» данной библиотеки. Данный пакет предоставляет API для взаимодействия с сервисом Amazon S3[6].

1.5 Операции с сервисом S3: «Download» и «Upload»

Пакет «service/s3» библиотеки AWS SDK для Go предоставляет два пакета, с одним из которых я непосредственно буду работать. Этим пакетом является «service/s3/s3manager», который предоставляет API для взаимодействия с Amazon S3. Двум рассматриваемым операциям взаимодействия с S3 соответствуют два объекта, доступных для создания – это «Uploader» и «Downloader». Рассматриваемый пакет «service/s3/s3manager» предоставляет ряд утилит для произведения операций «Upload» и «Download». В следующем параграфе более детально будет рассмотрен процесс создания объектов «Uploader» и «Downloader», а также будут написаны программы для совершения операций «Upload» и «Download».

1.6 Программы, реализующие операции «Download» и «Upload»

После ознакомления с пакетом «service/s3/s3manager», я приступил к выполнению одной из подготовительных задач – написанию программ для загрузки файла в Amazon S3 и для скачивания файла из Amazon S3 с помощью библиотеки AWS SDK для Go. Данные программы – это важный элемент этапа подготовки, так как основная часть их кода будет использоваться при решении дальнейших задач. Для начала рассмотрим общие для программ шаги.

Первым шагом необходимо создать объект из пакета «aws/session» – этот объект является структурой типа «Session»[7]. Данная структура предназначена для хранения конфигураций клиента (например, идентификационных ключей пользователя и идентификатора региона). Для создания объекта типа «Session» используется функция:

«func New (cfgs ...*aws.Config) *Session»

из пакета «aws/session», которая опционально принимает в качестве аргументов ссылки на объекты типа «Config» из пакета «aws» и возвращает ссылку на объект типа «Session». Если в качестве аргументов ничего не передается, то при создании объекта «Session» поиск идентификационных ключей пользователя и идентификатора региона, которые настроены по умолчанию, будет осуществляться по алгоритмам, описанным в параграфе «Дополнительная настройка AWS SDK для Go». Если же в процессе написания программы возникла необходимость в использовании каких-либо иных идентификационных ключей пользователя или идентификатора региона, отличных от тех, что настроены по умолчанию, то можно в создающую функцию передать ссылки на объекты типа «Config» из пакета «aws», содержащие необходимые для конкретной сессии параметры, включая идентификационные ключи пользователя или идентификатор региона. В таком случае при создании объекта типа «Session» поиску «параметров по умолчанию» будут подлежать только те параметры, которые не были явно переданы при создании данного объекта с помощью объектов типа «Config» из пакета «aws».

Предыдущий шаг (создание сессии) из текущего параграфа одинаков для обеих программ. Далее написание каждой из них будет рассмотрено отдельно. Для каждой программы будут проиллюстрированы схемы, которые описывают логику их работы и взаимосвязи между объектами, создаваемыми в коде.

Для начала рассмотрим операцию «Upload». Иллюстрация соответствующей программы приведена на рисунке 3.

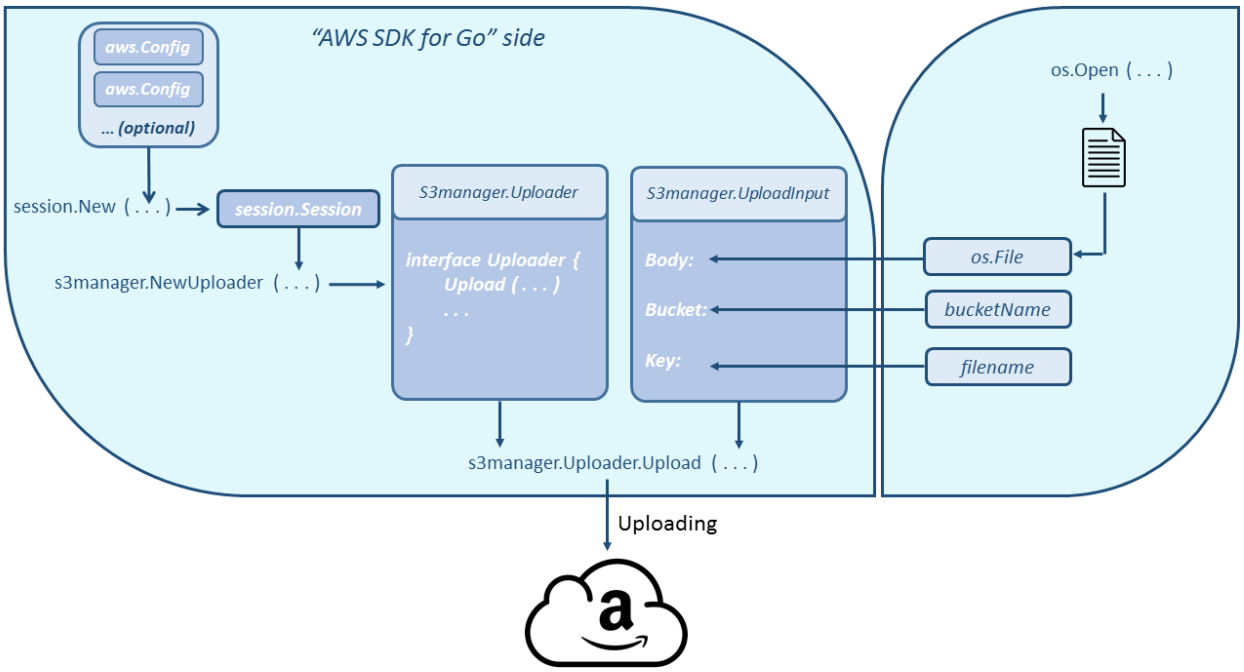

Рисунок 3 – Иллюстрация логики работы программы «Upload» и взаимосвязи объектов, создаваемых при написании программы

Для загрузки данных с помощью библиотеки AWS SDK для Go на Amazon S3 необходимо создать объект типа «service/s3/s3manager.Uploader» (далее «Uploader»)[8]. Данный объект реализует интерфейс, позволяющий производить загрузку данных на Amazon S3. Создание объекта типа «Uploader» производится вызовом функции:

«func NewUploader (c client.ConfigProvider, …) *Uploader»

из пакета «service/s3/s3manager». В качестве аргументов данная функция принимает объект типа «Session» из пакета «aws/session» – ранее созданную сессию и некоторые опциональные аргументы, которые я обозначил в виде троеточия. В данном примере и далее под троеточием будут подразумеваться опциональные аргументы, передача которых в функцию не производится. Для того, чтобы загрузить данные на Amazon S3, осталось вызвать функцию:

«func (u Uploader) Upload (input *UploadInput, …) (*UploadOutput, error)»,

которая является частью интерфейса, реализуемого объектом типа «Uploader». В качестве аргументов она принимает ссылку на объект типа «UploadInput» из пакета «service/s3/s3manager» и некоторые опциональные параметры. Таким образом, для загрузки данных на Amazon S3 осталось создать объект типа «UploadInput» и передать его в качестве аргумента в функцию загрузки. Объект «UploadInput» – это структура. При её создании будут явно инициализированы поля «Bucket», «Key» и «Body», остальные поля будут проинициализированы значениями по умолчанию. Поле «Bucket» – это строка, являющаяся именем целевого объекта «bucket», в котором будет храниться файл. Поле «Key» – это строка, являющаяся именем файла, который загружается в S3. Поле «Body» – это ссылка на объект типа «os.File». Данный объект будет создан при открытии того файла, который планируется загрузить на Amazon S3. В качестве финального шага перед загрузкой необходимо вызвать загружающую функцию, передав в нее в качестве аргумента созданный «UploadInput».

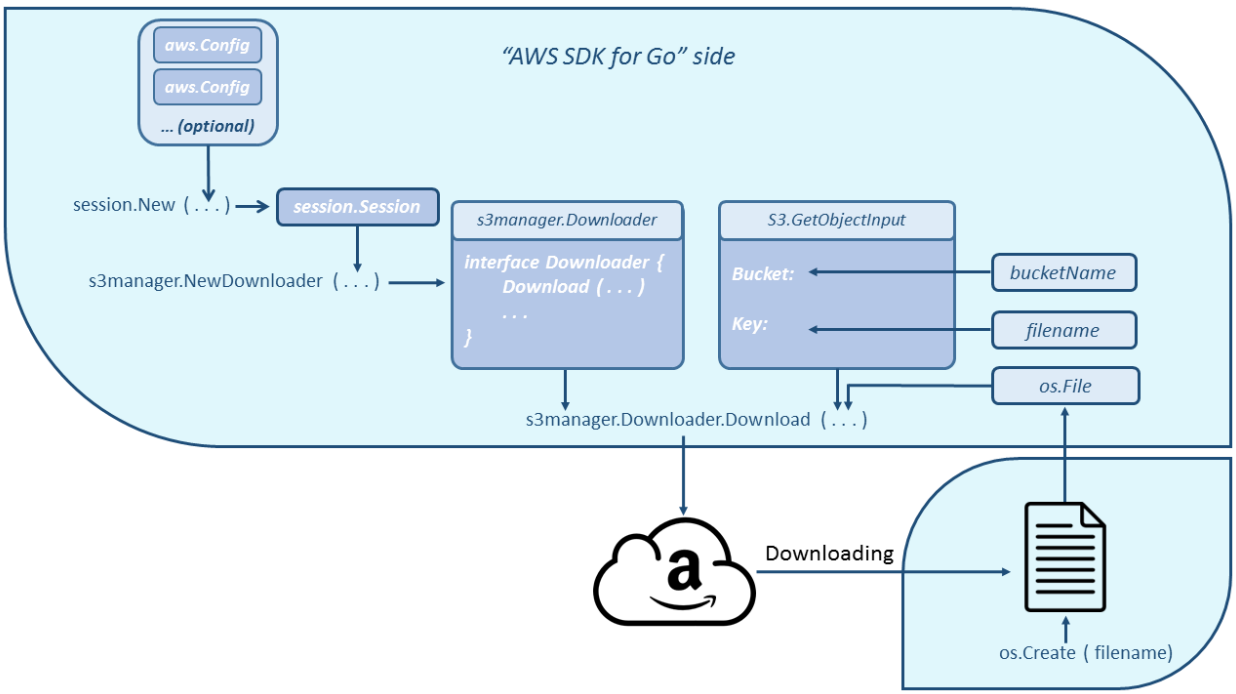

Рисунок 4 – Иллюстрация логики работы программы «Download» и взаимосвязи объектов, создаваемых при написании программы

Теперь рассмотрим операцию «Download»[8]. Иллюстрация соответствующей программы приведена на рисунке 4. Для загрузки данных с помощью библиотеки AWS SDK для Go из Amazon S3 необходимо создать объект типа «Downloader» из пакета «service/s3/s3manager». Данный объект реализует интерфейс, позволяющий производить загрузку данных из Amazon S3. Создание объекта типа «Downloader» производится вызовом функции:

«func NewDownloader (c client.ConfigProvider, …) *Downloader»

из пакета «service/s3/s3manager». В качестве аргументов данная функция принимает объекты того же типа, что и функция, которая использовалась для создания объекта типа «Uploader» из пакета «service/s3/s3manager». Загрузка данных из Amazon S3 производится с помощью функции:

«func (d Downloader) Download(w io.WriterAt, input *s3.GetObjectInput, …) (n, error)»,

которая является частью интерфейса, реализуемого объектом типа «Downloader». Данная функция в качестве аргументов принимает объект, реализующий интерфейс «io.WriterAt» (в данной работе я буду использовать для этих целей объект типа «os.File»), ссылку на объект типа «GetObjectInput» из пакета «service/s3» и некоторые опциональные аргументы. Создание объекта типа «os.File» происходит при создании файла, в который будут помещены загруженные из Amazon S3 данные. Осталось понять, что скрывает в себе объект типа «GetObjectInput».

Тип «GetObjectInput» – это структура. При её создании будут явно инициализированы поля «Bucket» и «Key» (данные поля имеют такой же смысл, как и в случае с «UploadInput» из пакета «service/s3/s3manager»), остальные поля будут проинициализированы значениями по умолчанию. Создав объект «GetObjectInput» и файл, в который будет производиться загрузка, можно переходить к загрузке данных из Amazon S3 вызовом загружающей функции.

1.7 Отладка взаимодействия с Amazon S3

Не маловажным является процесс отладки взаимодействия с Amazon S3. Если что-то происходит не так в процессе взаимодействия с сервисом, мало иметь возможность анализировать ошибки, возвращаемые библиотечными функциями. Имеет интерес возможность делать «dump» всех HTTP запросов и ответов. В данном параграфе будет рассмотрен механизм логгирования, предоставляемый библиотекой AWS SDK для Go. Также будет рассмотрен вопрос активации этого механизма, так как по умолчанию он выключен.

Пакетом «aws» предоставляется механизм логгирования. Найти его описание можно в файле «aws/logger.go»[9]. В данном файле объявлен интерфейс «Logger», который предполагает реализацию единственного метода «Log(… interface{})». В данном файле также реализован логгер, который по умолчанию используется библиотекой AWS SDK для Go. Данный логгер использует пакет «log»[10] стандартной библиотеки языка Go и все логи выводятся в «stdout». Стоит понять, как же его использовать. Для этого необходимо вновь вернуться к параграфу 6 данной главы. В нем обсуждался процесс написания программ, осуществляющих операции «Upload» и «Download». Первым делом создавался объект «Session» из пакета «aws/session»[7]. При его создании создающую функцию могут быть опционально переданы указатели на объекты типа «Config» из пакета «aws»[13]. Напомню, что объект данного типа - это структура, поля которой описывают конфигурации создаваемой сессии. Однако, ранее были рассмотрены только такие конфигурации, как идентификационные ключи пользователя или идентификатор региона. Все остальные конфигурации при создании объекта «Session» имели значение по умолчанию. Теперь для того, чтобы использовать логгер, предоставляемый библиотекой AWS SDK для Go, необходимо передать в создающую функцию в качестве аргумента ссылку на структуру

типа «Config» с явно инициализированным полем «LogLevel». Поле «LogLevel» отвечает за уровень логгирования. В файл «aws/logger.go»[9] объявлены константы, отвечающие различным уровням логгирования. По умолчанию при создании структуры «Config» значение поля «LogLevel» инициализируется в значение константы «LogOff» из пакета «aws», что означает, что механизм логгирования отключен. Для включения логгирования имеется ряд констант, причем одна из них в точности подходит для решения поставленной в этом параграфе задачи – «dump» всех HTTP запросов и ответов. Это константа «LogDebugWithHTTPBody» из пакета «aws». Инициализация поля «LogLevel» значением данной константы включит логгер, и он будет логгировать все HTTP запросы и ответы.

На данном этапе первый пункт постановки задачи может считаться выполненным. В процессе выполнения данного пункта я ознакомился с необходимыми мне для работы пакетами библиотеки AWS SDK для Go, что позволило написать программы, осуществляющие загрузку данных на Amazon S3 и загрузку данных из Amazon S3. Большая часть кода данных программ будет использоваться мной в дальнейшей работе. Пришло время перейти ко второму пункту постановки задачи и изучить возможность вызова функций, написанный на языке Go из кода, написанного на языке C.

Глава II. Вызов функций на Go из программ на C

2.1 Интеграция C и Go

В данном параграфе будет описан процесс написания двух программ на языке C, которые в процессе своей работы будут осуществлять вызовы функций, написанных на языке Go с использование библиотеки AWS SDK для Go. Одна из программ, написанных на языке C, будет осуществлять вызов функции, совершающей загрузку данных на Amazon S3, другая программа на C будет вызывать функцию, выполняющую обратную операцию – загрузку данных из Amazon S3.

Для начала необходимо разобраться с тем, какие инструменты предоставляет язык Go для того, чтобы код Go программ мог быть вызван из языка C. Начиная с версии языка Go 1.5, в инструмент «go build» был добавлен флаг «-buildmode»[11], который позволяет собирать Go пакеты в различных режимах. Всего их семь, но в данной работе меня интересуют только два из них - это: «-buildmode=c-archive» и «-buildmode=c-shared». Первый режим позволяет собирать Go «main» пакет и все импортируемые в него пакеты (например, пакеты из стандартной библиотеки языка Go, какие-либо пользовательские пакеты, такие как пакеты из AWS SDK для Go) в статическую библиотеку, которая может быть прилинкована к основной программе, написанной на C. При сборке с использование флага «-buildmode=c-shared» тот же самый код на Go будет собран уже в динамическую библиотеку. Таким образом появляется возможность вызывать из кода на C функций, написанных на языке Go. В данной главе я разберу детали реализации такого подхода. Оба значения флага представляют интерес, так как они позволяют получить ответ на поставленный в задаче вопрос.

При сборке с рассматриваемыми флагами имеются некоторые ограничения на собираемый пакет. Во-первых, собираемый пакет должен быть «package main». Во-вторых, важно, чтобы наряду с теми функциями, которые будут вызываться из C, также присутствовала функция «main», однако она должна иметь пустое тело, так как при сборке она игнорируется.

В своей работе я создал два файла: «get.go» и «put.go». Из названий этих файлов явно следует предназначение содержимого этих файлов. В файле «get.go» я описал функцию «get_go (…)», тело которой в точности соответствует телу функции, совершавшей операцию «Download» в прошлой главе. Соответственно, в файле «put.go» описана функция «put_go (…)», тело которой в точности повторяет тело функции, совершавшей операцию «Upload» в прошлой главе. Я разберу устройство файла «get.go», так как все шаги идентичны для обоих файлов.

Как уже было сказано, пакет должен быть «package main». Далее импортируются все необходимые сторонние пакеты (пакеты из стандартной библиотеки языка Go и пакеты из библиотеки AWS SDK для Go). После этого необходимо импортировать псевдо-пакет «C», выглядит это так: «import “C”»[12]. На данном этапе необходимо понять, что же это за псевдо-пакет и какая от него польза. Если посмотреть на стандартную библиотеку языка Go, то станет очевидно, что в ней нет такого пакета. Именно поэтому он называется псевдо-пакетом. Импортирование данного псевдо-пакета очень важно. “C” – это имя, которое интерпретируется «cgo» как некая отсылка к пространству имен языка C. Без импортирования данного псевдо-пакета невозможно произвести сборку пакета с флагами «-buildmode=c-archive» и «-buildmode=c-shared». Далее в файле «get.go» я описал функцию «get_go (…)». Для того, чтобы функцию «get_go (…)» можно было вызвать из программы на C, необходимо использовать директиву «//export» до объявления функции. Общая структура файла «get.go» проиллюстрирована на рисунке 5:

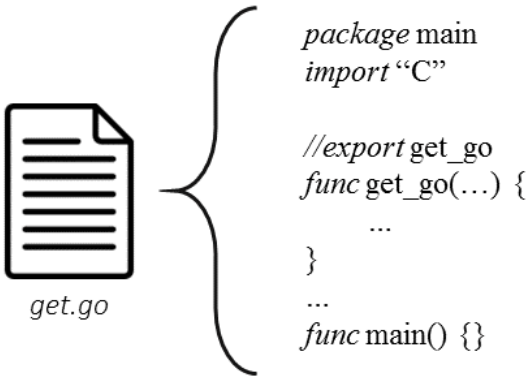

Рисунок 5 – Содержимое файла «get.go»

Далее необходимо не забыть включить в пакет функцию «func main()» с пустым телом. Это необходимо для того, чтобы рассматриваемый пакет (package main) был собран, однако, наличие функции «func main()» не предполагает того, что эта функция будет точкой входа в программу. Точкой входа будет функция «main» программы на C.

Далее можно приступать к сборке с помощью инструмента «go build», используя при сборке флаг «-buildmode=c-archive» или «-buildmode=c-shared». На рисунке проиллюстрирован результат сборки с флагом «-buildmode=c-shared»:

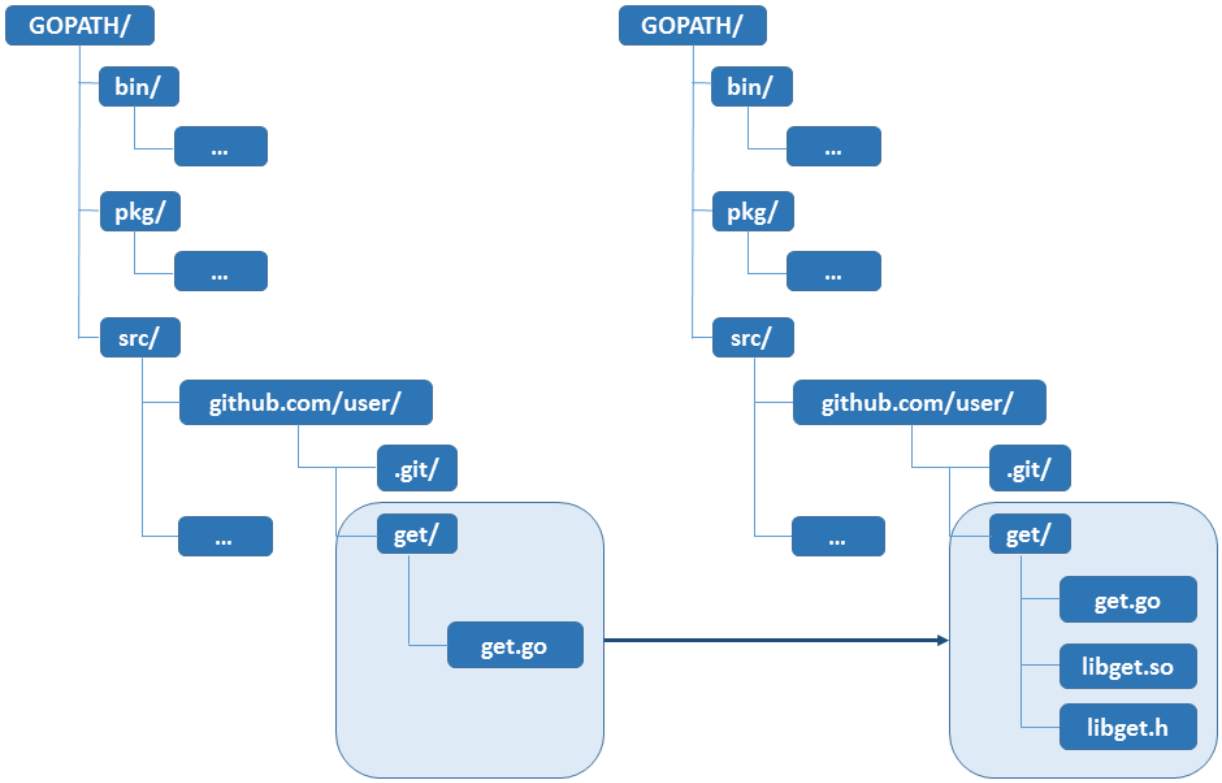

Рисунок 6 – Иллюстрация результатов работы инструмента «go build» с флагом «-buildmode=c-shared»

Если сборку производить со значением флага «-buildmode=c-archive», то результат будет отличаться лишь заменой файла «libget.so» на «libget.a».

В программе, написанной на C, в функции «main» производится вызов функции «get_go». Чтобы компиляция программы прошла успешно, важно не забыть включить в программу заголовочный файл «libget.h», полученный при сборке Go кода. Запуск полученного после компиляции исполняемого файла, как и ожидалось, производит загрузку файла на Amazon S3. На данном этапе имя загружаемого на Amazon S3 файла было прописано в коде функции «get_go», что, очевидно, не приемлемо, так как есть потребность производить загрузку произвольного файла, что потребует при подобном подходе вносить изменения в исходный код «get.go» и затем производить повторную сборку.

Имя файла должно быть каким-либо образом передано в загружающую функцию, которая находится на стороне Go кода. И передаваться это имя должно со стороны C кода, поэтому на следующем этапе необходимо понять то, как передавать аргументы из кода на C в функцию, написанную на языке Go.

2.2 Вызов Go кода из C с передачей элементарных аргументов

Для начала стоит понять, как передавать различные элементарные аргументы, допустим какое-либо целое число, то есть тип «int». Перед тем, как описывать изменения, которые необходимо произвести в коде программы, стоит вспомнить, что при написании «put.go» и «get.go» в коде импортировался псевдо-пакет “C”. Было сказано, что этот псевдо-пакет с именем “C” является некой отсылкой в пространство имён языка C. То есть тип «int» со стороны Go будет выглядеть как «C.int»[14].

Таким образом, для того, чтобы можно было передать целочисленное значение из кода, написанного на C, в функцию, написанную на Go, необходимо внести некоторые изменения в уже написанный в прошлом параграфе код. Во-первых, со стороны программы на C при вызове функции «get_go (…)» или «put_go (…)» надо не забыть передать в качестве аргумента какое-либо целочисленное значение. Со стороны Go программы в качестве принимаемого функцией «put_go (…)» или «get_go (…)» аргумента необходимо указать аргумент типа «C.int»:

«func put_go (n C.int)»,

«func get_go (n C.int)».

Пример с типом «int» является показательным, по аналогии можно передавать такие типы данных, как: char (C.char), unsigned char (C.uchar), short (C.short),

unsigned short (C.ushort), unsigned int (C.uint), long (C.long), unsigned long (C.ulong), long long (C.longlong), unsigned long long (C.ulonglong), float (C.float), double (C.double). Однако, в данной работе помимо передачи аргументов вышеперечисленных типов, понадобится передать указатель на структуру и строку. Тем самым, необходимо разобраться с тем, как это сделать[14].

2.3 Передача строк и структур в качестве аргументов функции

В языке C, как известно, нет строкового типа. Строка представляется в виде массива символов. Концом строки является специальный символ конца строки. Встает вопрос: как передать строку в функцию на Go? Со стороны кода, написанного на C, передача подобного аргумента будет выглядеть как передача указателя на начало строки. Со стороны языка Go, в качестве принимаемого аргумента должен быть аргумент типа «*C.char»[14]. Объявление функций будет выглядеть следующим образом:

«func put_go (str *C.char)»,

«func get_go (str *C.char)».

В данной работе в функции «put_go (…)» и «get_go (…)» в качестве аргументов будут передаваться две строки. Это становится понятно, если вернуться к рисункам 3 и 4. В обоих случаях при создании структур «UploadInput» и «GetObjectInput» явно инициализируются поля «Bucket» и «Key» данных структур. Первое поле имеет тип «string», оно несет в себе имя целевого «bucket». Второе поле – это имя хранимого объекта, оно тоже имеет тип «string». Именно поэтому передавать необходимо две строки. После передачи строк, ими необходимо воспользоваться на стороне программы, написанной на языке Go. Для этого нужно уметь конвертировать тип «*C.char»

в строковый тип языка Go. Данная операция может быть проведена с помощью функции:

«func C.GoString (*C.char) string»,

которая, как видно, принимает в качестве аргумента тип «*C.char» и возвращает объект типа «string»[14].

Полученные после конвертации строки можно использовать при инициализации полей структур «UploadInput» и «GetObjectInput».

В следующей главе понадобится передать в качестве одного из аргументов функции «put_go (…)» и «get_go (…)» указатель на структуру, поэтому важно понять, как производить такую передачу. Для начала стоит разобраться с тем, как передать структуру. Допустим, на стороне программы, написанной на C, имеется структура «struct str {…}». Тогда при объявлении функций «put_go (…)» и «get_go (…)» в качестве принимаемого аргумента должен указываться тип «C.struct_str», где «C.struct_» обозначает то, что в качестве аргумента передается объект, являющийся структурой языка C, а после указывается название структуры, то есть «str»[14].

«func put_go (str C.struct_str)»,

«func get_go (str C.struct_str)».

Тип указателя на структуру отличается лишь добавлением символа “*”:

«func put_go (str *C.struct_str)»,

«func get_go (str *C.struct_str)».

В данной главе был дан ответ на второй пункт постановки задачи. Была доказана возможность вызова функции, написанной на языке Go из программы, написанной на языке C. В таких вызовах интерес представляет возможность передачи аргументов в вызываемую функцию. Поэтому было

показано, как передавать элементарные аргументы, а также строки и указатели на структуры – это полный набор необходимых в данной работе типов данных, передаваемых в функцию, написанную на языке Go, из программы, написанной на языке C.

Глава III. Работа с библиотекой «libpcs_io»

3.1 Введение к главе

Содержание данной главы – пожалуй, самая главная часть работы. На этом этапе будут использованы все ранее полученные результаты. Данная глава полностью покрывает третий пункт из раздела «Постановка задачи», в ней будут продемонстрированы два способа организации неблокирующего вызова Go кода из «libpcs_io-корутин». Также в конце главы будет произведено обоснование того, почему обработка сигналов будет происходить бесконфликтно.

3.2 Неблокирующий вызов Go кода из «libpcs_io-корутины»

Деталей работы «libpcs_io» я описывать не стану. Стоит лишь отметить, что в основе работы данной библиотеки лежит цикл событий (event loop). Данный цикл событий оперирует корутинами. Корутина – это компонента программы, обобщающая понятие подпрограммы, которая дополнительно поддерживает множество входных точек, остановку и продолжение выполнения с сохранением определенного контекста. Код корутины предполагает наложение на него нескольких ограничений. Одно из них состоит в том, что код должен быть неблокирующим. Все блокирующие операции необходимо выгружать в отдельный системный поток, отличный от потока цикла событий. О другом ограничении речь пойдет в следующем параграфе.

Работа с файловой системой может рассматриваться как операция, требующая ожидания. То есть необходим механизм, позволяющий выгружать операции подобного рода в отдельный системный поток, отличный от потока цикла событий. Такой механизм реализован в «libpcs_io», он называется «filejob». Рассмотрим пример программы с использованием данного механизма. Пусть имеются два файла. Один из файлов имеет имя «infile», второй – «outfile». Необходимо написать программу, в которой будут создаваться две корутины: одна корутина будет читать данные из файла «infile» и записать их в «pipe». Для этой операции в программе будет реализована функция «reader», указатель на которую будет использован при создании соответствующей корутины. Вторая корутина будет читать из «pipe» и писать прочитанные данные в файл «outfile». Для этой операции также будет реализована функция «writer», указатель на которую будет использоваться при создании второй корутины. Помимо создания корутин также необходимо создать и проинициализировать два объекта «rfj» и «wfj», отвечающих за механизм «filejob», соответственно для операций «read» и «write».

Как было написано в прошлом абзаце, первая корутина выполняет код функции «reader», вторая корутина – код функции «writer». Стоит описать логику работы обеих функций. Начну с функции «reader». Схема ее работы продемонстрирована на рисунке 7. Словесное описание работы данной программы представлено после рисунка.

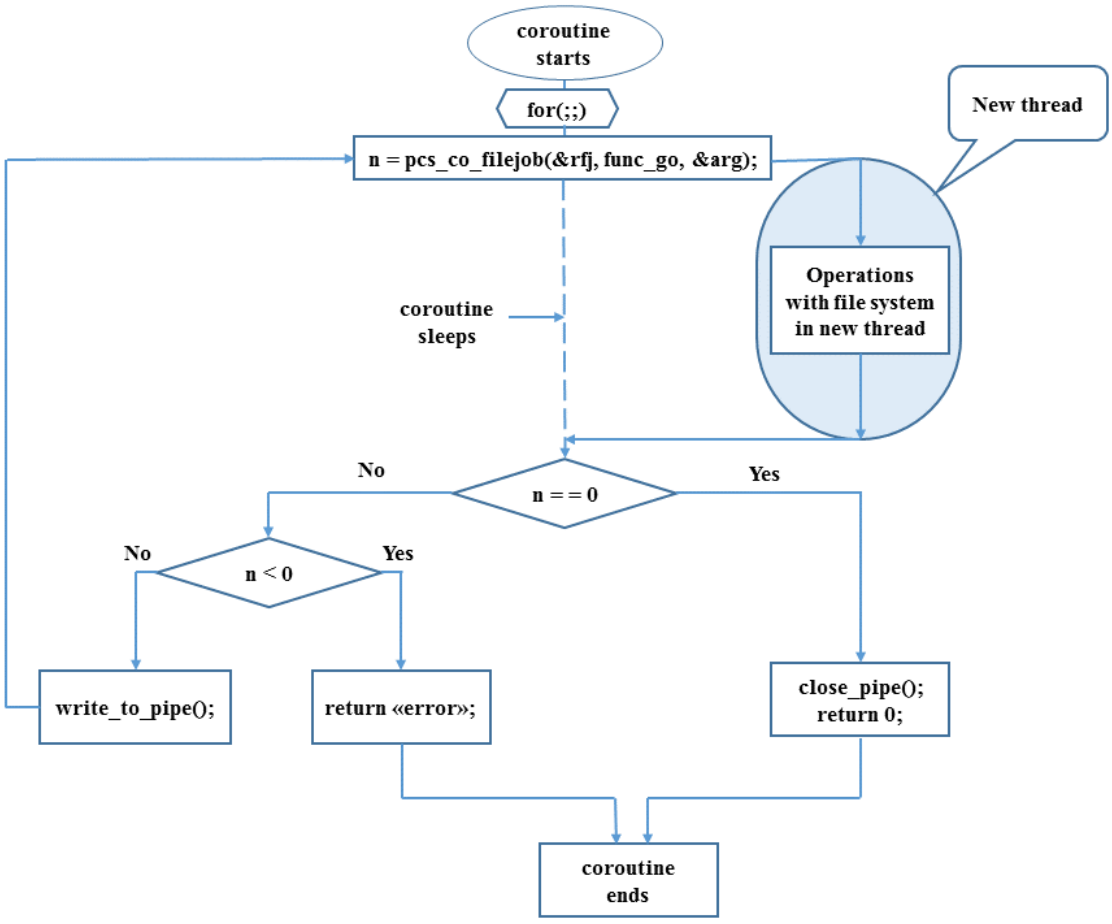

Рисунок 7 – Иллюстрация логики работы корутины, производящей чтение из файла и запись в «pipe»

Тело функции – бесконечный цикл, который прерывается тогда, когда файл полностью прочитан. Опишем пошагово работу программы. Первым делом происходит чтение из файла. Для этого вызывается функция «pcs_co_filejob (&rfj, read_foo, &arg)», в которую передается указатель на «rfj», указатель на функцию, осуществляющую чтение из файла, и указатель на структуру, описывающую буфер, в который надо записать прочитанные из файла данные. Вызов функции «pcs_co_filejob(…)» влечет за собой ряд последствий. Во-первых, происходит создание нового системного потока, в который производится выгрузка задачи чтения из файла. Во-вторых, для того, чтобы корутина не ждала окончания работы вновь созданного потока, она

засыпает, что фактически говорит о том, что может проснуться вторая корутина. После завершения чтения из файла происходит вызов функции, которая будит спящую корутину, и корутина вновь получает управление. Работа «pcs_co_filejob(...)» подразумевает возврат целого числа, которое интерпретируется как число байт, прочитанных из файла. Если число байт равно нулю, то закрывается соответствующий файловый дескриптор «pipe» и программа завершается с кодом возврата 0. Выполнение данной ветки говорит о том, что программа отработала без ошибок, файл полностью прочитан и все его содержимое записано в «pipe». Если же возвращаемое значение меньше нуля, то при чтении файла произошла ошибка, что влечет к возврату из функции «reader» с кодом ошибки. Если же возвращаемое значение больше нуля, то в данной ситуации произошла запись прочитанных данных из файла в «pipe» и после этого начинается новая итерация. Цикл итерирует до тех пор, пока в файле имеются непрочитанные данные. На следующем рисунке проиллюстрирована схема работы функции «writer». Описывать схему не имеет смысла, так как логически «writer» и «reader» устроены одинаково с точки зрения работы инструмента «filejob». На рисунке 8 проиллюстрирована логика работы корутины, производящей чтение из «pipe» и запись в целевой файл. Данный пример программы, в работе которой участвуют две корутины, реализующие логики, проиллюстрированные на рисунках 7 и 8, демонстрирует работу механизма, именуемого «filejob».

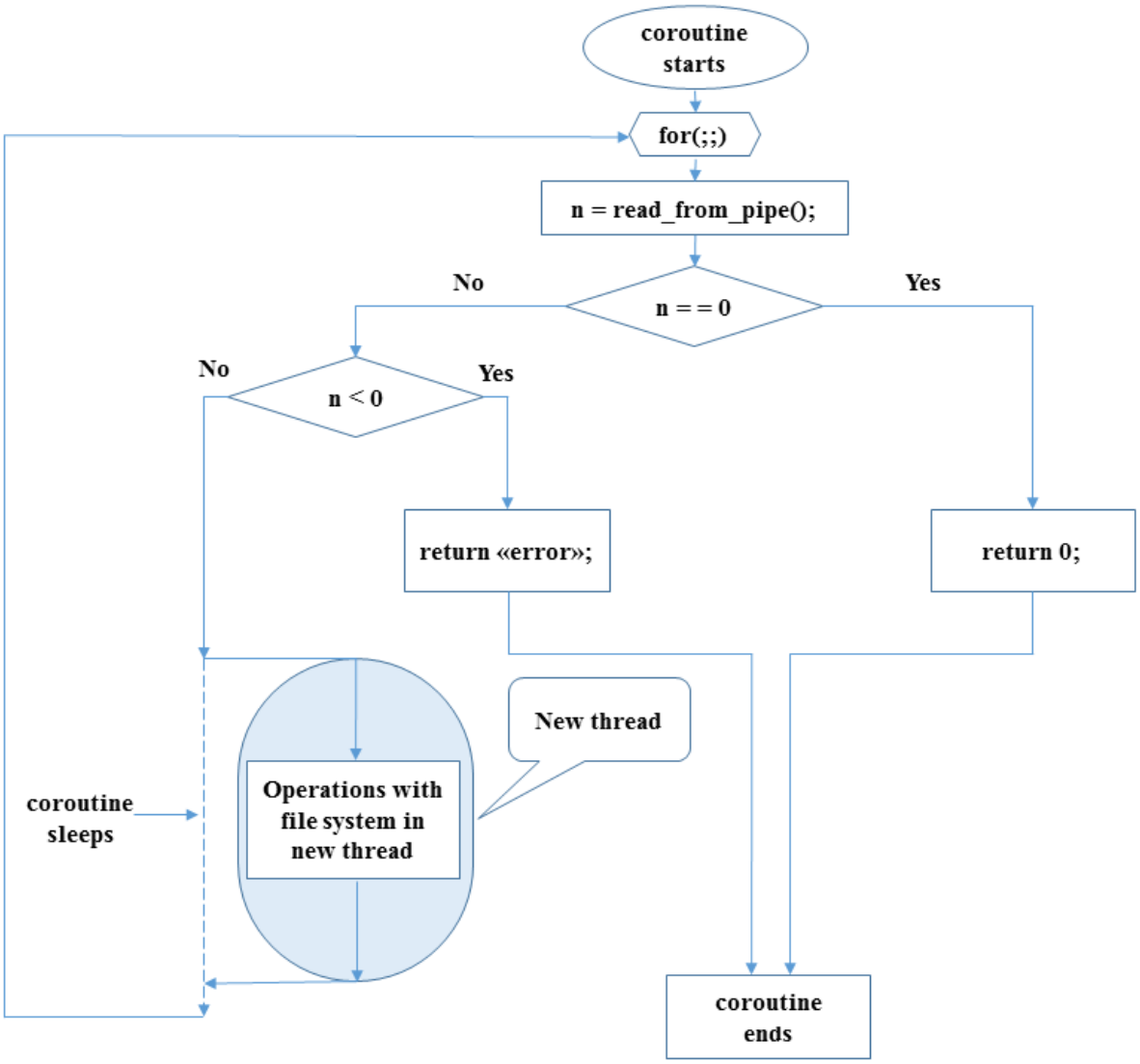

Рисунок 8 – Иллюстрация логики работы корутины, производящей чтение из «pipe» и запись прочитанных данных в файл

Разобравшись с «filejob», можно приступать к решению поставленной задачи. Очевидно, что решение будет опираться на механизм «filejob». По аналогии с предыдущим примером будут созданы «gfj» - объект, отвечающий реализации механизма «filejob», «go_caller» - функция, указатель на которую используется при создании корутины и «func_go» - это именно та функция, которая выполняет работу на стороне Go, и указатель на эту функцию будет передаваться при вызове функции «pcs_co_filejob(&gfj, func_go, NULL)»

наряду с указателем на объект «gfj» и NULL в качестве указателя на структуру, описывающую буфер (в данном случае он нам не нужен). На рисунке 9 проиллюстрирована логика работы корутины, отвечающей за вызов Go функции.

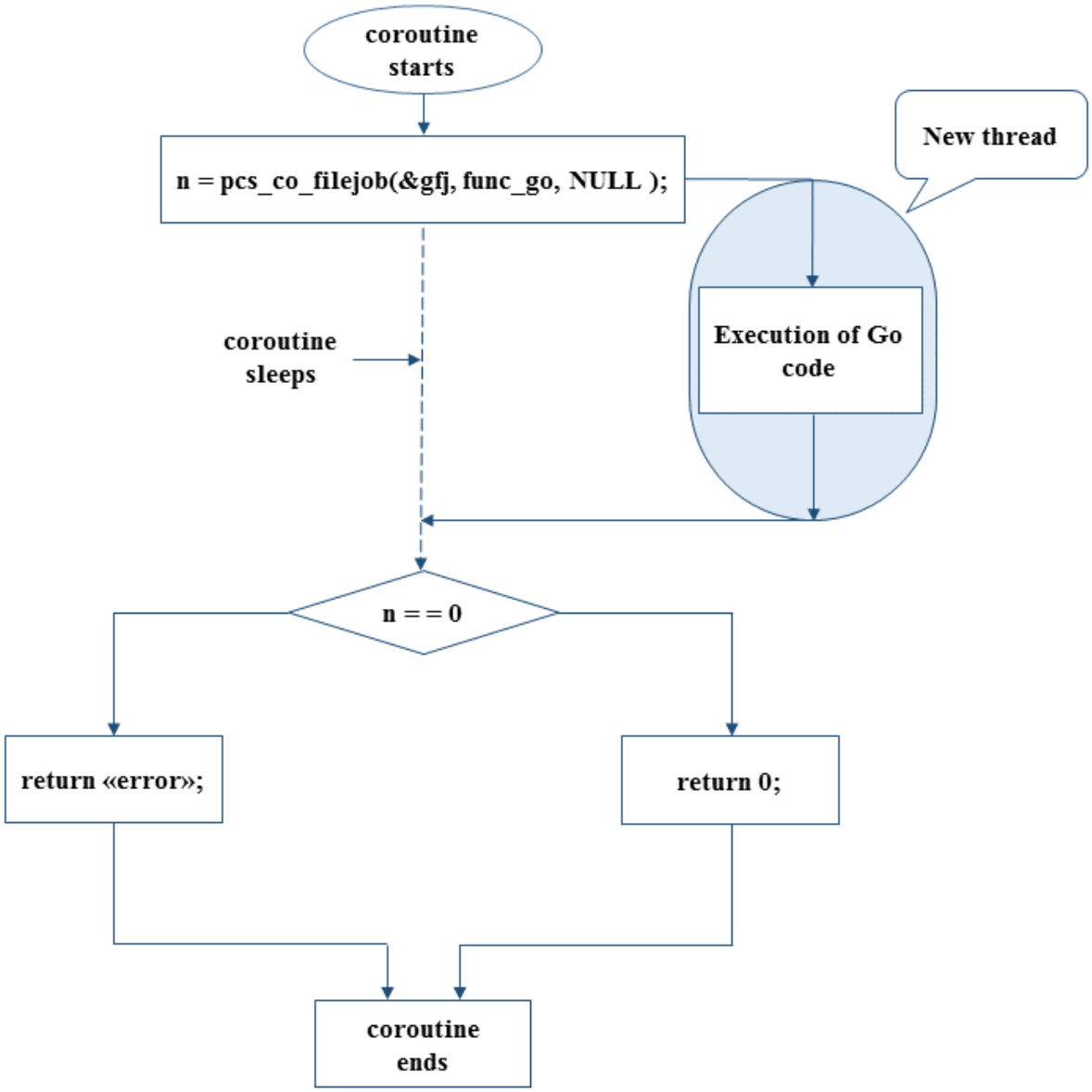

Рисунок 9 – Иллюстрация логики работы корутины, производящей вызов функции, написанной на языке Go с использованием механизма «filejob»

В данном случае логика функции «go_caller» очень проста. В теле данной функции имеется лишь вызов функции «pcs_co_filejob(...)» с передачей

в нее вышеописанных аргументов. Этот вызов дает возможность создать отдельный системный поток и заставить корутину заснуть на время выполнения кода функции «func_go». Замечу, что далее в работе я буду упоминать только функцию «func_go». Под «func_go» могут подразумеваться ранее написанные функции «put_go» и «get_go». В данной главе важно разработать общий механизм выгрузки этих операций в отдельный системный поток без заострения внимания на каждой из них. Запуски написанной программы позволяют утверждать, что данный подход довольно успешен. Однако, в последнем примере выгрузка производилась в отдельный системный поток, и это происходило на стороне C. Можно ли сделать такую же выгрузку операций работы с файловой системой на стороне Go? На стороне Go, в отличие от “C”, будет возможность оперировать не системными потоками, а горутинами. Как известно, горутина более легковесна, чем системный поток с точки зрения ее создания и переключения контекста при работе программы. Легковесность горутин сказывается и на их численности. Go runtime поддерживает работу числа горутин, которое имеет порядок миллиона. В случае с системными потоками такой многочисленности достичь невозможно. Все вышеперечисленные преимущества наталкивают на мысль о том, чтобы попробовать реализовать механизм выгрузки задачи работы с файловой системой на стороне Go. Решение данной задачи я представлю в два этапа. В первом этапе имеются недоработки, которые исправляются во втором этапе.

Для решения поставленной задачи, как и в прошлых примерах, создается корутина. В данном случае создаваемая корутина будет выполнять непосредственно код функции «func_go». Но код этой функции придется немного изменить. Отличия продемонстрированы на рисунке 10.

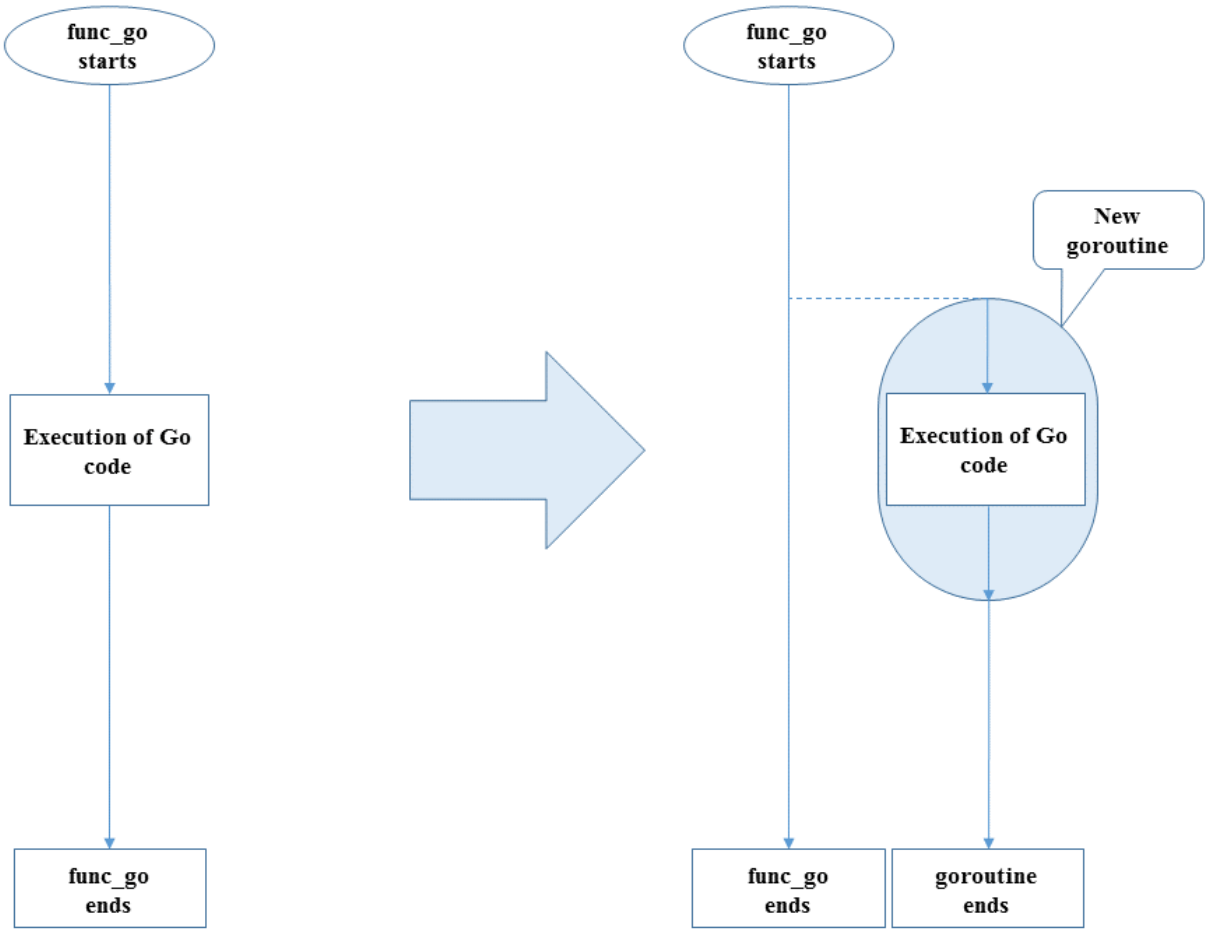

Рисунок 10 – Иллюстрация произведенных изменений в логике работы функции «func_go»

При данной логике работы функции «func_go», логика работы корутины, выполняющей работу с файловой системой, будет выглядеть так, как показано на рисунке 11. Ранее функция «func_go» в своем теле выполняла лишь необходимые операции с файловой системой и завершалась. Для решения поставленной задачи в новой версии функции происходит создание горутины, которой делегируется выполнение операций над файловой системой. Сама программа, не дожидаясь окончания выполнения горутины,

завершается. Данный пример является удачным в плане работоспособности. Все необходимые операции над файловой системой завершаются успешно.

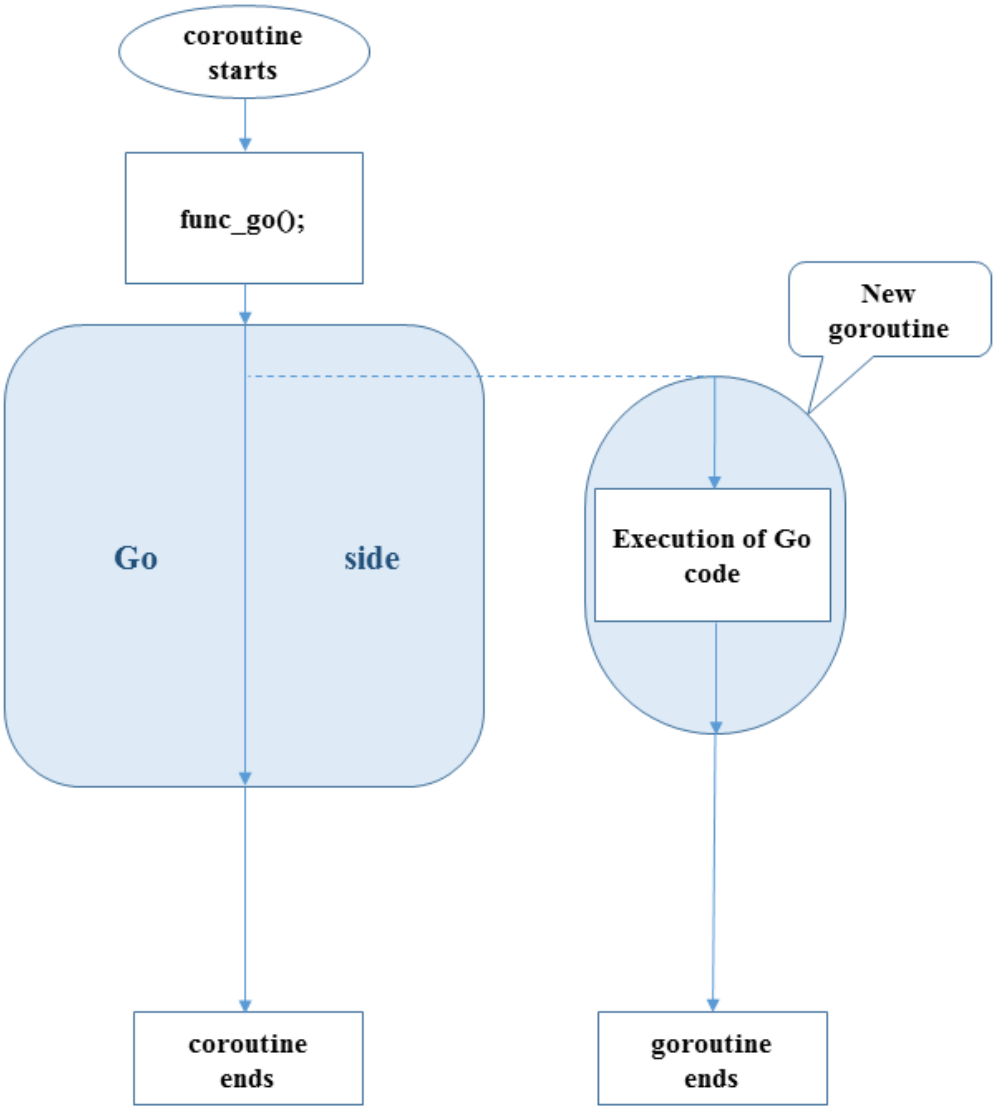

Рисунок 11 – Логика работы корутины, производящей вызов функции «func_go», которая в свою очередь создает новую горутину

Однако, как я писал ранее, код корутины – это код функции «func_go». Данная функция завершается раньше, чем происходит завершение работы горутины, которой делегируются операции над файловой системой. То есть сама корутина завершается раньше, чем завершается та задача, которая была перед ней поставлена. В этом и есть недостаток описанного сценария, а точнее

сказать – его недоработка. Следующее решение имеет более сложную форму, но оно исправляет ту недоработку, с которой пришлось столкнуться в прошлом примере. При создании корутины я снова использую функцию «go_caller», но ее тело также претерпевает некоторые изменения. На следующей иллюстрации продемонстрирована логика работы функции «go_caller».

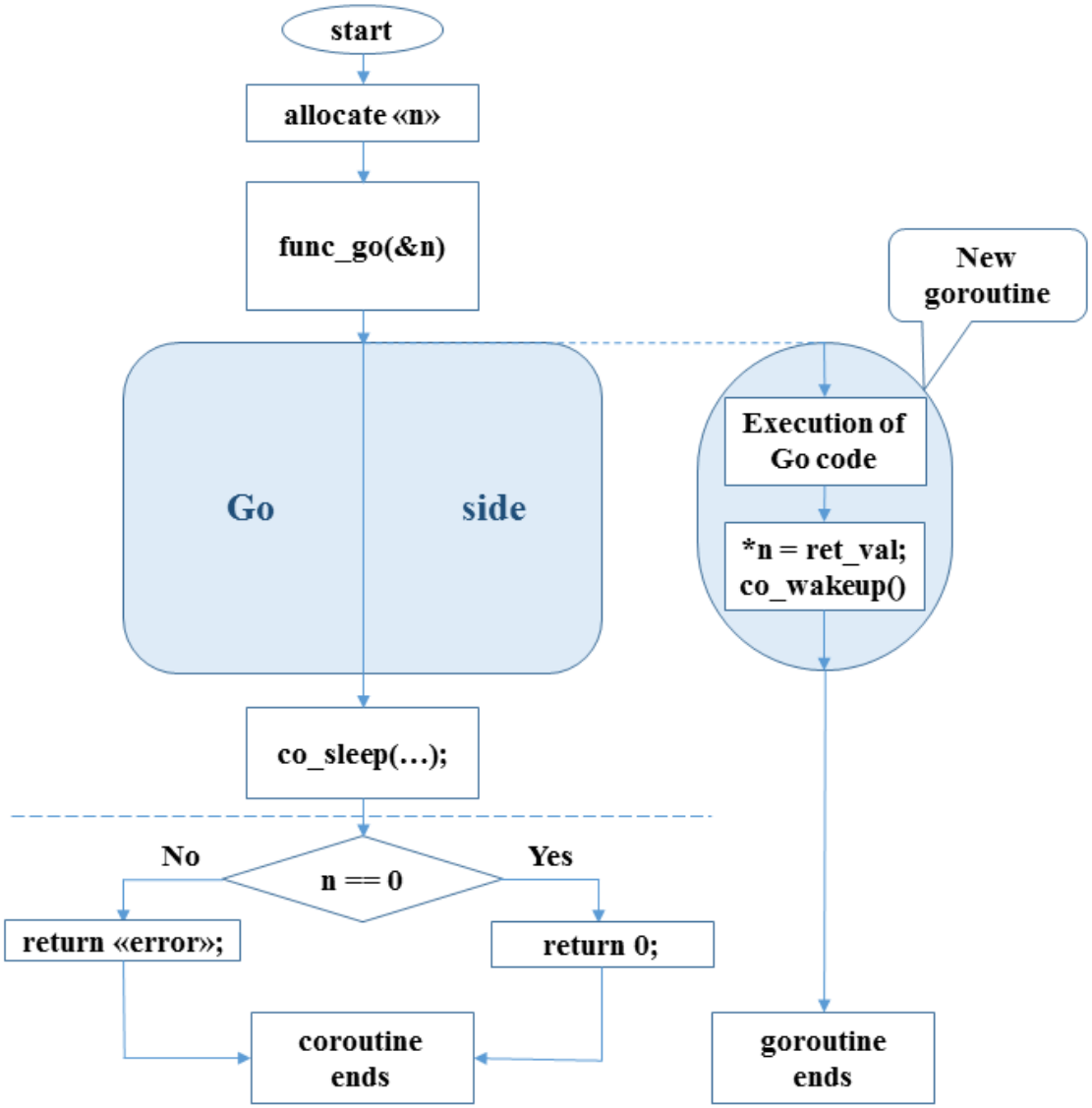

Рисунок 12 – Иллюстрация работы корутины, дожидающейся окончания работы горутины

В начале выделяется память под некий объект, который будет являться возвращаемым значением. Далее происходит вызов функции «func_go» с

передачей ей в качестве аргумента того участка памяти, который был выделен на первом шаге. После вызова «func_go» мы попадаем в Go runtime. Тело данной функции уже знакомо, оно было ранее проиллюстрировано. В этой функции производится создание горутины с передачей ей в качестве аргумента того указателя на память, который был передан как аргумент в функцию «func_go». После создания горутины «func_go» завершается, не дожидаясь завершения горутины. Мы вновь попадаем в функцию «go_caller», в которой производится вызов функции «pcs_co_sleep(&co…)", реализованной в «libpcs_io», что заставляет корутину заснуть. В качестве одного из аргументов данная функция принимает указатель на объект, являющийся структурой, описывающей корутину. Именно для этого ранее рассматривался способ передачи указателя на структуру в качестве аргумента при вызове функции, написанной на Go, из кода C. Единственным объектом, который может разбудить корутину в нужный момент, является горутина, созданная при выполнении функции «func_go». Данной горутине придется выполнить вызов функции «pcs_co_wakeup(...)», которая также реализована в «libpcs_io». Встает задача, обратная задаче из второй главы: как из Go вызвать C?

Данный вопрос решается довольно просто. Необходимо соблюдение некоторых правил написания кода в файле, в котором описана функция «func_go». Пусть этот файл именуется как «func.go», тогда его структура будет выглядеть так, как показано на рисунке 13.

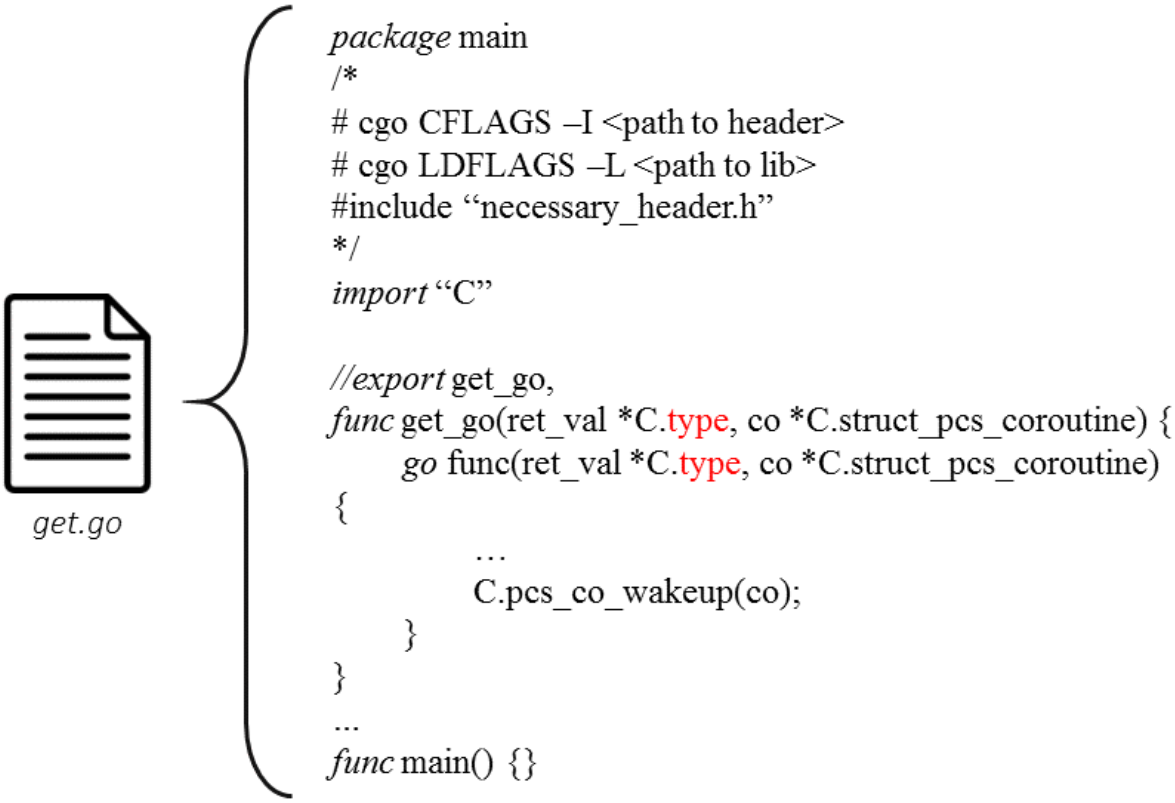

Рисунок 13 – Общая структура программы в файле «get.go»

Ранее уже была продемонстрирована структура файла с кодом на Go для загрузки из Amazon S3. Однако, для решения возникшей задачи вызова C из кода на Go, появилась необходимость в дополнении файла определенным содержимым. Перед «import “C”» появилась закомментированная область. Ранее заходила речь об инструменте «cgo». «Cgo» распознает эту закомментированную область. Любая строка, начинающаяся с “#cgo” с последующим знаком пробела удаляется из комментария. Данные метки являются директивами для «cgo». Оставшиеся строчки представляют из себя набор директив «#include ...», которые используются при компиляции части пакета, которая написана на C. В нашем случае следует включить заголовочный файл, в котором определена функция «pcs_co_wakeup(...)». После проделанных изменений становится возможно произвести вызов функции «pcs_co_wakeup(...)» из горутины. Перед вызовом «pcs_co_wakeup(...)» горутина выполняет делегированную ей работу и

сохраняет результат своей работы в ту область памяти, указатель на которую был передан ей в качестве аргумента. Это означает, что горутина выполнила поставленную перед ней задачу и может благополучно завершиться. Перед завершением она производит вызов «pcs_co_wakeup(...)», что пробуждает корутину. После этого вызова горутина завершает свою работу. Пробужденная корутина проверяет возвращенное значение на предмет наличия ошибки при работе горутины и завершает свое выполнение. Оба подхода являются решением поставленной задачи. Однако, на мой взгляд, все же лучшим из них является подход с использованием «filejob». Если еще раз вернуться к рисунку с иллюстрацией работы второго подхода (с выгрузкой задачи в горутину), то может возникнуть вопрос об очередности выполнения вызовов «pcs_co_sleep(...)» и «pcs_co_wakeup(...)». Теоретически может случиться так, что вызов второй функции произойдет раньше вызова первой. Но на практике (на тестовых программах) такого не происходило.

3.3 Бесконфликтная работа окружений «libpcs_io» и Go

Другое ограничение, накладываемое на код корутины касается бесконфликтности работы кода Go и «libpcs_io». В «libpcs_io» устанавливаются свои обработчики сигналов. Так как в Go тоже имеются механизмы обработки сигналов, то необходимо обеспечить бесконфликтную работу обеих сторон. Для решения данного вопроса стоит обратиться к официальной документации языка Go, а точнее, к описанию пакета «os/signal». В описании данного пакета есть раздел, посвященный рассматриваемой проблеме. Разработчики языка Go позаботились о том, чтобы обеспечить бесконфликтность в обработке сигналов. Далее я кратко опишу основные положения, приведенные в официальной документации.

В случае, если производить сборку пакета с помощью команды «go build -buildmode=c-shared» (собственно, это рассматриваемый в данной работе

случай), то инициализация Go runtime (соответственно и установка обработчиков сигналов) происходит в момент, когда загружается библиотека (libget.so/libput.so). Теперь подробности того, что происходит[15]:

1) Если Go runtime видит уже имеющиеся у C кода обработчики сигналов для «SIGCANCEL» и «SIGSETXID», то для этих сигналов будет применен флаг «SA_ONSTACK» и будет сохранен обработчик со стороны C кода.

2) Для «synchronous» сигналов Go runtime установит обработчики в любом случае, но применение будет таким (также предполагается, что для подобных сигналов со стороны C кода имеются обработчики): если сигнал поступает во время исполнения C кода, то в обработку данного сигнала вовлечется обработчик со стороны C кода вместо обработчика со стороны Go (в случае, если он есть). В противном случае вызовется Go обработчик (Go обработчики выставляются по умолчанию для «synchronous» сигналов).

3) Для всех остальных сигналов (в случае сборки в режиме «-buildmode=c-shared») Go-runtime не станет устанавливать никакие обработчики по умолчанию. Обработчики сигналов либо устанавливаются вручную, либо используются уже установленные обработчики со стороны C кода. Опишу подробнее: если со стороны C кода имеются свои обработчики сигналов рассматриваемого типа, то для этих сигналов Go runtime применит флаг «SA_ONSTACK» и сохранит обработчик со стороны C кода**.** Если для какого-то «asynchronous» сигнала будет необходимо установить обработчик со стороны Go, то необходимо вызвать функцию «Notify()» из пакета «os/signal» для данного сигнала, тогда Go runtime установить свой обработчик. Если затем будет вызвана функция «Reset()» (отменить ранее выполненные с помощью «Notify» изменения для данного сигнала) из того же пакета, то первоначальный обработчик сигнала (обработчик со стороны C кода) будет установлен в качестве обработчика по умолчанию, если он имелся.

В данной главе были рассмотрены два подхода к организации неблокирующего вызова Go кода из «libpcs_io – корутины». Один из них использовал реализованный в «libpcs_io» механизм, именуемый «filejob». Второй механизм был связан с созданием дополнительной горутины для выгрузки в нее задачи, осуществляющей работу с файловой системой. Оба механизма полностью решают поставленную задачу, однако первый подход, связанный с использованием механизма «filejob», выглядит более надежным. Что касается обработки сигналов, то из описания, предоставленного в данной главе, можно сделать вывод, что разработчики языка Go позаботились о возможных проблемах, связанных с конфликтами при обработке сигналов и реализовали решение, которое обеспечивает бесконфликтную работу C и Go с точки зрения обработки сигналов.

Глава IV. Отладка смеси C и Go

4.1 Отладка Go с помощью GDB и LLDB

Написание программ, особенно сложных, сопровождается ошибками. Необходимо иметь возможность обнаруживать, локализовать и устранять возникшие ошибки, проще говоря, отлаживать программу. Отладку можно производить несколькими способами. Один из них – использование программ-отладчиков. Другой способ – вывод текущего состояния программы с помощью расположенных в критических точках операторов вывода. Оба способа будут рассмотрены в этой главе. Однако, начать стоит с программ-отладчиков.

Для процесса отладки существуют различные программы-отладчики. В своей работе я рассматривал GDB и LLDB[16][17]. Для начала стоит сформулировать общую проблему обеих программ-отладчиков. Данная проблема заключается в том, что и GDB, и LLDB не понимают принципа работы Go runtime. Однако, разработчики обеих программ-отладчиков попытались сделать поддержку языка Go. Получилось это сделать не у всех. У LLDB гораздо больше нерешенных вопросов, и расширение, которое предоставляется данным отладчиком, имеет менее широкую функциональность, чем GDB. Поэтому основное внимание я уделю GDB.

Начать стоит со сборки пакета с программой. Сборка будет проводиться с помощью инструмента «go build». Для успешной работы отладчика необходимо при сборке не забыть передать несколько флагов[16][17]:

«go build –gcflags “-N -l” prog.go»

Применение данных флагов очень важно при сборке пакета для отладки. Дело все в том, что компилятор языка Go по умолчанию производит некоторые

оптимизации кода, такие, например, как встраивание функций, оптимизирует работу с переменными. Все производимые компилятором оптимизации попросту затрудняют отладку кода. Для отключения оптимизаций необходимо использовать указанные в примере флаги.

Для отладки кода Go с помощью GDB было написано расширение, которое можно найти в исходниках языка Go. Это скрипт, написанный на языке Python. При установке языка Go в «/usr/local», путь до расширения будет выглядеть так: «/usr/local/go/runtime/runtime-gdb.py»[16]. Данное расширение подгружается с помощью команды «source» с указанием полного пути до файла с расширением (GDB позволяет загружать расширения). Данное расширение добавляет в GDB набор команд, осуществляющих поддержку таких объектов языка Go, как «goroutine», «string», «slice», «map», «channel» или «interface»[18]. Описывать работу с каждым из объектов не имеет смысла, стоит лишь сформулировать общее впечатление. На мой взгляд, отладка с помощью GDB – не лучший путь, даже в ситуации, когда имеется поддержка языка Go. Очень сложно представить отладку кода в случае, когда приходится иметь дело с огромным количеством горутин. Язык Go так устроен, что их численность может достигать миллиона. И инспектировать отладчиком текущее состояние такой конкурентной среды – довольно сложная задача. Однако, есть одна важная проблема, которая делает отладку кода, написанного на языке Go, с помощью GDB совершенно не привлекательной. Речь об этой проблеме пойдет в следующем параграфе.

4.2 Отладка смеси C и Go с помощью GDB

Когда приходится отлаживать смесь C и Go, то в этой ситуации возникает проблема. Ранее в главе 2 было написано, что для того, чтобы использовать код, написанный на языке Go, из программ, написанных на языке

C, необходимо произвести сборку пакета в динамическую библиотеку. Такая сборка осуществляется инструментом «go build» следующим образом:

«go build –buildmode=c-shared prog.go»

Однако, из написанного в предыдущем параграфе следует, что при сборке пакета стоит также использовать дополнительные флаги, которые отключают оптимизации, производимые компилятором по умолчанию. В результате команда для сборки будет иметь следующий вид:

«go build –gcflags “-N -l” –buildmode=c-shared prog.go»

Такая сборка заканчивается ошибкой. Решение данной проблемы мне не удалось найти. Без флагов, отключающих оптимизацию, производимую компилятором по умолчанию, отлаживать часть, написанную на Go, становится гораздо сложнее.

Отладка смеси C и Go происходит так же, как если бы производилась отладка кода на C, производящего вызов функции из какой-либо динамической библиотеки. Разница лишь в том, при нахождении в Go-части следует пользоваться командами, предоставляемыми расширением, чтобы корректно интерпретировать детали работы Go runtime (Go-объекты различных типов). Однако, из написанного в начале данного параграфа следует, что отладка смеси C и Go с помощью GDB становится совершенно не привлекательной из-за возникновения проблем со сборкой программы с отключенными оптимизациями. В следующем параграфе будут кратко рассмотрены механизм, которые предоставляются самим языком Go для отладки кода.

4.3 Использование встроенных механизмов отладки в Go

Помимо использования программ-отладчиков, можно пользоваться механизмами, позволяющими инспектировать текущее состояние программы с помощью отладочной печати. Для отладочной печати прекрасно подойдет пакет «log» – это стандартной пакет языка Go. Для того, чтобы увидеть детали работы Go runtime, можно воспользоваться пакетами «runtime» и «runtime/debug»[19][20]. В них реализованы методы, позволяющие инспектировать текущие состояния горутин (например, содержимое стека горутины), позволяют вывести общее число исполняемых горутин, позволяют узнавать некоторые параметры Go runtime, такие, как число системных потоков, размер стека, статистику работы с памятью (garbage collection) и многое другое. Пакеты «runtime» и «runtime/debug» позволяют получать достоверную информацию о работе Go runtime. Это свойство является очень ценным, если сравнивать данный подход к отладке программы с тем подходом, в котором использовались программы-отладчики. При отладке кода, написанного на Go, с помощью GDB или LLDB нельзя полностью быть уверенным в достоверности того, что выводит отладчик.

В данной главе был дан ответ на четвертый пункт постановки задачи. В ней были рассмотрены несколько подходов к отладке кода, написанного на языке Go (также смеси C и Go). Первый подход – использование программ-отладчиков. Данный подход не очень хорош в случае с языком Go, так как поведение Go runtime отличается от того поведения, которое ожидают от программы рассматриваемые отладчики. Второй подход – использование механизмов, предоставляемых языком Go. Данный подход наиболее привлекателен ввиду того, что данные механизмы демонстрируют достоверную информацию о «жизни» Go runtime. Касательно смеси C и Go: отлаживать часть на C можно с использованием программ-отладчиков, а

отладка Go-части может производиться механизмами, предоставляемыми пакетами «log»[10], «runtime» и «debug». Такой подход, на мой взгляд, наиболее эффективен.

Список сокращений и терминов

1. Бекап – процесс создания копии данных на носителе, предназначенном для восстановления данных в оригинальном или новом месте их расположения в случае их повреждения или разрушения[21].
2. Amazon S3 (от англ. Amazon Simple Storage Service) – название одного из сервисов, предоставляемых компанией Amazon[22].
3. SDK (от англ. Software Development Kit) – набор средств разработки, который позволяет специалистам по программному обеспечению создавать приложения для определённого пакета программ, программного обеспечения базовых средств разработки, аппаратной платформы, компьютерной системы, игровых консолей, операционных систем и прочих платформ[23].
4. Горутина (англ. goroutine) – средство организации многопоточности в языке Go[24].
5. Корутина (англ. coroutine) - компонент программы, обобщающий понятие подпрограммы, который дополнительно поддерживает множество входных точек (а не одну как подпрограмма), остановку и продолжение выполнения с сохранением определённого положения[25].

Выводы

В данном разделе будут подведены итоги проделанной мной работы. В первой главе была проделана работа по ознакомлению с языком Go, настройкой рабочего пространства языка Go, было произведено ознакомление с библиотекой AWS SDK для Go, также были написаны две тестовые программы, осуществляющие взаимодействие с сервисом Amazon S3. Предназначение данных программ – выполнение операций «Upload» и «Download». Написание этих программ важно для решения последующих задач. Не маловажным является поиск способов отладки взаимодействия с сервисом S3, в процессе исследования библиотеки был найден способ логгировать все HTTP запросы и ответы. Это позволит видеть причину проблем при их возникновении. Стандартные механизмы обработки ошибок могут попросту не корректно интерпретировать ошибки, возникающие в процессе взаимодействия с сервисом. Работа, проделанная в первой главе, дает исчерпывающий ответ на первый пункт из раздела «Постановка задачи».

Во второй главе была проведена работа по исследованию возможности осуществлять вызовы функций, написанных на языке Go, из кода, написанного на языке C. В данной главе были представлены методы сборки пакетов, которые делают возможным осуществлять подобные вызовы. Но мало интереса представляет возможность вызова функции без передачи в нее необходимых аргументов. Поэтому в рассматриваемой главе также был продемонстрирован способ передачи аргументов при вызове функции, написанной на языке Go, из кода, написанного на языке C. В данной работе передаваемыми аргументами были только указатели на структуры и строки. Результаты этой главы полностью решают задачу из второго пункта раздела «Постановка задачи».

В третьей главе была проделана работа по использованию функций, написанных на языке Go из «libpcs_io - корутин». Были рассмотрены два метода, реализующие выгрузку Go кода в отдельный поток исполнения, отличный от потока цикла событий. В первом примере выгрузка производилась в отдельный системный поток и занимался этим вопросом механизм, реализованный в «libpcs_io», именуемый «filejob». Однако, не менее интересен подход, реализующий возможность выгрузки задачи в отдельный поток на стороне Go. Этот механизм связан с созданием горутины, которой будет делегирована задача. Оба подхода работоспособны. Однако, реализовав взаимодействие «libpcs_io» с кодом, написанным на языке Go, необходимо убедиться в том, что не будут возникать конфликты, связанные с обработкой сигналов, поступающих во время работы программы. Разработчики языка Go позаботились о том, чтобы при сборке в рассматриваемых в работе режимах, не было возникновения подобных конфликтов. Таким образом, в рассматриваемой главе полностью решается задача, поставленная в третьем пункте из раздела «Постановка задачи».

В четвертой главе обсуждалась возможность отладки смеси C и Go с помощью программ-отладчиков GDB и LLDB. Стоит сказать сразу, что процесс отладки кода, написанного на языке Go, с помощью рассматриваемых программ-отладчиков довольно затруднителен, даже при наличии расширений у обеих программ. Отладка усложняется, когда приходится отлаживать смесь C и Go, так как не удается отключить оптимизации, которые по умолчанию выполняются компилятором языка Go. Однако, отлаживать программу на Go и смесь C и Go можно, используя имеющиеся в языке Go пакеты «runtime» и «runtime/debug». В данных пакетах реализованы функции, позволяющие инспектировать состояние Go runtime. Проделанная в данной главе работа покрывает четвертый пункт из раздела «Постановка задачи».

Источники

1. Официальный сайт языка Go // https://golang.org/
2. Установка языка Go на Linux // https://sllite.ru/2016/03/установка-последней-версии-go-на-ubuntu/
3. Исходный код AWS SDK для Go // https://github.com/aws/aws-sdk-go
4. Руководство по устройству рабочего пространства // https://golang.org/doc/code.html
5. Дополнительная настройка AWS SDK для Go // https://docs.aws.amazon.com/sdk-for-go/v1/developer-guide/configuring-sdk.html
6. Пакет для работы с сервисом S3 // https://github.com/aws/aws-sdk-go/tree/master/service/s3
7. Документация по пакету «aws/session» // http://docs.aws.amazon.com/sdk-for-go/api/aws/session/
8. Документация по пакету «service/s3/s3manager» // http://docs.aws.amazon.com/sdk-for-go/api/service/s3/s3manager/
9. Исходный код логгера, реализованного в AWS SDK для Go // https://github.com/aws/aws-sdk-go/blob/master/aws/logger.go
10. Исходный код пакета «log» языка Go // https://golang.org/pkg/log/
11. Режимы сборки пакетов Go // https://docs.google.com/document/d/1nr-TQHw_er6GOQRsF6T43GGhFDelrAP0NqSS_00RgZQ/edit#
12. Описание инструмента cgo // https://blog.golang.org/c-go-cgo
13. Описание пакета «aws» // https://github.com/aws/aws-sdk-go/tree/master/aws

14. Передача аргументов при вызове функций, написанный на языке Go, из кода, написанного на языке C // https://golang.org/cmd/cgo/
15. Официальная документация по пакету «os/signal» языка Go // https://golang.org/pkg/os/signal/
16. Отладка языка Go с помощью программы-отладчика GDB // https://blog.codeship.com/using-gdb-debugger-with-go/ https://golang.org/doc/gdb
17. Отладка языка с помощью программы-отладчика LLDB // http://ribrdb.github.io/lldb/
18. Типы данных в Go // http://golang-book.ru/
19. Документация по пакету «runtime» // https://golang.org/pkg/runtime/
20. Документация по пакету «runtime/debug» // https://golang.org/pkg/runtime/debug/
21. Определение понятия «бекап» // https://ru.wikipedia.org/wiki/Резервное_копирование
22. Официальный сайт сервиса Amazon S3 // https://aws.amazon.com/ru/s3/
23. Определение понятия SDK // https://ru.wikipedia.org/wiki/SDK
24. Средства для организации многопоточности в языке Go // http://golang-book.ru/chapter-10-concurrency.html
25. Определение понятия «корутина» // https://ru.wikipedia.org/wiki/Сопрограмма