И, напоследок, давайте поговорим про сигналы, которые мы можем получить внутри контейнера или про системные сигналы Linux, поскольку сигналы внутри контейнера, которые мы получаем, ничем не будут отличаться от обычных системных сигналов Linux-а.

Значит, мы знаем, что любое приложение от системы может получить уведомления о том, что система думает что-то о приложении. Соответственно, может произойти простой сигнал в приложение о том, что он употребляет слишком много процессора или у системы не хватает памяти, или, что приложение сейчас будет выключено. Оно не просто может, оно, в принципе, посылается. Не все приложения это отрабатывают.

В Go это реализовано путём создания специального канала channel-а, который ловит специальные сигналы. И соответственно, их отслушивает. Мы уже видели этот код в, соответственно, предыдущей лекции, в лекции, где я показывал HTTP-сервер, он там был для обработки, так называемого, graceful shutdown. Для того, чтобы обработать процесс завершения сервера. Повторюсь, здесь os.Signal и Signal.Notify – это, соответственно, наши друзья в обработке и в получении…, и обработке сигналов. Соответственно, сигналы со стороны docker-а могут посылаться в приложение любые, с помощью команды docker Kill вы можете послать, в принципе, любой код в приложение. И посмотреть, как оно себя ведёт, если вам это важно, соответственно, обрабатывать.

Но, сам golang гарантирует нам, что получить точно мы можем только 2 сигнала – это interrupt и Kill. И опять же, получить точно – это не значит, что мы их гарантированно получим. Потому что, например, сигнал Kill в большинстве Linux-систем не ждёт никакой реакции от приложения, он сразу его завершает. То есть у нас для совместимости со всеми системами гарантируется только два типа сигнала. Опять же, если вы знаете, что в вашей системе есть больше сигналов, вы

можете их спокойно ловить самостоятельно. И, в случае нештатной ситуации, docker выключает контейнер с сигналом sigkill. И перехватить его, соответственно, нельзя. Мы сейчас посмотрим, как это работает.

Соответственно, из тех сигналов, которые приходят системой, можно выделить следующие типы сигналов – sigint, сигнал отключения от пользователя, тот самый interrupt, который мы, вполне себе, можем готовить в приложение и как-то поработать. Это сигнал, так называемый, мягкий, который система посылает, когда мы хотим выключить приложение, и приложение может решить, что же ему делать с этим фактом. Может там затребовать дополнительное время, сохранить все свои данные, записать что-то на диск, а потом спокойно выключиться. Sigkill нельзя перехватить. Это у нас наша команда Kill -9 в Linux-е. Это так называемая (НРЗ 03:10), когда мы говорим, что в приложении что-то совсем не так, оно больно, и его просто надо убить. В таких случаях просто убивается процесс.

Sigterm посылается системой процессу. Это, с точки зрения процесса…, это, по идеи, то же самое, что и sigint, что interrupt. Но посылается он не пользователем, а системой. Например, система обнаружила нехватку каких-то ресурсов и хочет это приложение, поскольку оно неважное, убить. Но убить мягко, потому что ресурсы всё ещё есть, то есть, можно дать приложению время подумать и закончить. Соответственно, я знаю, что в некоторых реализациях, система (НРЗ 03:53) в случае превышения ресурсов, посылается сначала sigterm, а потом sigkill. То есть, у вас есть шансы обработать сигнал прерывания перед тем, как ваше приложение будет убито.

И ещё один из сигналов, который я хотел бы рассмотреть сегодня, это сигнал sigxcpu. Это такой специфический сигнал, его, наверное, сложно встретить в повседневной жизни какого-то там Web Deploy-я. Значит, что процесс использует слишком много времени процессора, то есть превысил лимиты по процессору. На

системах Openshift, которые у нас были в компании развернуты, стоял, соответственно, контролер, который посылал именно вот эту команду, потому что наши приложения могли себя внутри скалировать. То есть могли меньше выполнять расчётов для того, чтобы забирать меньше времени процессора. Насколько я понимаю, в Kubernetes-ах и в системах, с которыми вы будете работать, скорее всего этот сигнал у вас появляться будет примерно никогда, но такой интересный сигнал. Ну и, конечно, это такие сигналы, которые я выделил, и их на самом деле, довольно-таки много. И есть спецификации Linux-а, но активно они никакими системами (НРЗ 05:18), на данный момент не используются. Только, если вы сами, соответственно, его напишите.

Давайте посмотрим, как сигналы работают на практике. Итак, я подготовил тестовый проект. Что у нас здесь происходит. У нас здесь генерируется строка вот такой вот длины, и эта строка раз в секунду у нас копируется, причём максимально небезопасным способом. То есть здесь я избегаю всех вариаций оптимизации компилятора, когда он там может увидеть, что строка одинаковая. И значит, будет копировать ссылку на строку каждый раз. Здесь мы этого избегаем для того, чтобы сэмулировать утечки памяти. И таким образом показать прерывание docker-контейнера по памяти, при убивании docker контейнера по памяти.

Соответственно, что здесь у нас есть. У нас как раз есть наш channel. Мы создаём channel специального типа os.Signal, который представляет собой специальный тип для оборачивания сигналов системы операционной. И с помощью команды Signal.Notify, мы передаём этот канал. И указываем сигналы, которые могут здесь быть. Опять же, как я уже говорил, golang гарантирует, что у нас в системе будет interrupt и kill. Опять же, гарантирует – это такое интересное слово, потому что мы знаем, что kill у нас в приложении может и не прокинуться.

И, если вы хотите туда ещё добавить сигналов специфичной для вашей системы, вы можете это сделать, например, через пакет Cisco для Linux-а, который в себе содержит основные сигналы SIGHUB, SIGINT, SIGPIPE, ALARM, SIGTERM и так далее. То есть те, которые…, вот TERM должны быть вам уже знаком. Ну и соответственно, SIGKill тоже здесь есть, но он замаскирован, соответственно, пакетом os.Kill. Видите, здесь просто константа.

Соответственно, мы здесь получаем вот этот сигнал. И эту горутину…, вернее, этот канал мы обрабатываем в отдельной горутине. Здесь мы можем получить вот этот как раз Signal, строковые его описания. И сам сигнал мы можем, собственно говоря, его напечатать и посмотреть, что же там нам пришло. Дальше, получая сигнал, я вываливаюсь с кодом ошибки. И соответственно, прерываю выполнение цикла. Давайте соберём наш контейнер, наш docker-файл. Здесь он собран по образу и подобию того docker-файла, который я показывал вначале build и run. Не знаю, зачем я здесь выставлял порт 8080, у меня здесь ничего нет.

Давайте соберём контейнер, запустим его и посмотрим, как работает вот это вот конструкция. Так вот, я собрал image для контейнера. Значит, image у меня называется myserv, собрал его 13 минут назад почему-то, не знаю почему. В общем, давайте запустим этот контейнер с помощью команды docker run -d myserv. -d отсоединит вывод docker-а. И позволит нам, соответственно…, выведет просто лишние контейнера и завершит процесс.

Соответственно, второй способ смотреть логи, если вы это делаете локально, это зайти в контейнер…, вернее, (НРЗ 08:49). Зайти вот сюда. И здесь у вас будут выводиться логи, и здесь вы можете посмотреть, что там в контейнере есть. Вот можно видеть, что от памяти постоянно отжигает всё больше и больше. Ну, а здесь можно увидеть логи. Единственный лог, который у меня есть, это Sistem call. Давайте вызовем команду docker Kill с префиксом Signal. И допустим, передадим

Signal=1. Signal=1 – это Signal hangup, это значит в нормальной жизни, что с приложением потеряна связь, значит, повешена трубка. И у него код 1, который мы сюда и передаем. И передаем, значит, ID-шник контейнера. Контейнер завершил свою работу, и мы здесь увидели, что Sistem call у нас, соответственно, hangup, он к нам приехал.

Второй код, который мы можем посмотреть – это код Sigint, опять же, user interrupted, гораздо более часто используемый. Не настолько искусственные, как тот, который я только, что передал, вот этот Signal hangup, я давно не видел, чтобы приложения таким образом убивались. Возможно, в вашей практике это иначе. Соответственно, второй сигнал – это Sigint, здесь мы, опять же, передаём, заходим в логи и видим Sistem call Interrupt. А, вот теперь давайте посмотрим, что получится, если мы передадим туда сигнал 9. Передаём сигнал 9, соответственно, запускаем наш сервер и убиваем его с сигналом 9. Но, перед тем как мы его убьём, я хочу вывести его логи.

Опа, выход с кодом ошибки 137, (НРЗ 10:34), так сказать. Завершился контейнер у нас аварийно, и ничего не написал в логе, то, о чём я как раз говорил. Сигнал 9 не дает никакого времени приложению, ни на какую обработку. То есть у вас, в принципе, оно будет завершатся всегда аварийно. И такое вполне возможно, например, docker, при превышении памяти, убивает контейнер самостоятельно, хотя козалось бы, да. И соответственно, вы это никак отловить не можете, Kubernetes немножко себя немножко лучше ведёт в некоторых сборках. Он присылает сначала Sigint или Signal Man и ждёт вашей реакции. Если реакции нету, приложение продолжает отжирать, он его убивает.

Давайте посмотрим, как это работает. Просто, так сказать, для общей эрудиции. Для того, чтобы проиллюстрировать данный эффект, напоминаю, что мы генерируем рандомную строчку. И начинаем её копировать, постоянно

клонировать в наш slice. То есть в какой-то момент наш slice может вырасти до довольно больших значений. И запускаем мы наш контейнер с ограничением по памяти. Здесь я поставил, наверное, 32 мегабайта. И с ограничением по memory swap. Почему? Потому что что может такое случиться, особенно, если вы хотите достичь этого эффекта-убийство по памяти, что docker будет использовать swap машина. То есть выделить машине какое-то место на диске и продолжить работать с приложением.

Соответственно, здесь я жёстко ограничиваю swap также, как и память. И запускаю наш сервис. Его можно увидеть здесь. Опять же, в логах ничего, здесь мы видим статистику по памяти. Соответственно, она сейчас у нас будет расти совместно с копированием нашей строчки в golang-е раз в секунду. И мы можем сейчас увидеть, прямо сейчас, как произойдёт убийство контейнера. То есть docker в этом плане не является джентльменом, как видите, он убивает контейнер по коду 137 без всяких логов. То есть прямо просто его берёт и выключает. Опять же, во многих системах облачных будьте также готовы, что ваше приложение, при недостатке памяти…, тоже с ними особо никто церемониться не будет, они просто умрут и будут загружаться заново.