Ну и базовый хрестоматийный пример — это чтение из файла. Давайте посмотрим, как это работает. Соответственно, у меня здесь есть функция error, которая просто паникует, когда какая-то есть ошибка, я её просто использую для того, чтобы обрабатывать ошибки от файлов. И есть один метод os.ReadFile, из которого я потом читаю и печатаю данные. Соответственно, читается это все из файла test.txt, который лежит в одной папке. Соответственно можно сюда и поставить полный путь, тоже будет читаться. Соответственно, что он здесь написано? Здесь написан test. И есть перенос строки. Если мы сейчас запустим нашу функцию main, то увидим – вот он, тот самый вывод.
Давайте посмотрим, как работает этот os.ReadFile. То есть она выгружает данные из filepath, полностью читая файл, вгружает их в наш dat, в массив байтов dat. Соответственно, мы видим, что функция ReadFile возвращает массив байтов, массив байтов — это единственное представление, которое вы можете получить из чтения файла, на самом деле, потому что файл не всегда является, как таковой, строкой. Иногда он является какой-то, вернее, он всегда может являться строкой, но, как правило, последовательность byte, который вы можете уже свободно преобразовать в другие типы данных. Соответственно, что здесь происходит? Здесь получается, файл из функции open. Open файл на чтение.

И соответственно, после этого, если ошибка и существует, мы возвращаем её. И если нет, то, соответственно, объявляем, что файл до выхода из функции надо закрыть. Получаем его размер с помощью функции, встроенный файл функции stat. Соответственно, здесь вот как раз уже начинается платформозависимая история. В Linux-е и Windows-е, соответственно, статистика, по файлу хранится по-разному, мы получаем размер файла, мы добавляем один байт для end of file, для специального символа, который закрывает файл. Если размер, соответственно, меньше 512 байт, то мы все равно читаем 512 байт, потому что в Linux-е существует костыль, некоторые файлы могут показывать ноль, но, тем не менее в себе что-то содержать.

И делаем массив байтов, читаем этот массив байтов из файла до, соответственно, конца файла, и его просто возвращаем. И закрываем. То есть эта функция прочитает все за раз и вернёт в память. Круто, как, казалось бы. А что ещё нужно? Но нужно, на самом деле, больше, потому что этот файл может быть очень большим, и в память он вам может полностью не влезть. И известный факт, что диски больше, чем оперативная память по размеру на большинстве машин. И, соответственно, читать огромный файл прям сразу в памяти, это не лучшая идея. Поэтому существуют другие методы чтения файлов.

Нам очень часто нужно не просто считать какие-то данные с файла, их обработать и всё. Нам очень часто нужно считать данные кусочками, так называемыми, Chunk-ами. Например, если у нас огромный CSV-файл или огромный какой-то конфигурационный файл, который нам не обязательно читать за раз, мы можем прочитать там первую часть, прочитать вторую часть и так далее.

Конечно, есть такая возможность во всех языках программирования, и в GO она тоже есть, она реализована, в принципе, почти как везде. Мы открываем, значит, файл по пути. Опять же, возвращается нам файл, это та же самая функция open, которая у нас была уже рассмотрена в первом примере. Соответственно, и дальше мы можем читать файл кусочками. Ну, что я делаю здесь? Здесь, я, например, делал массив байтов, размером 7 байт, и говорю, что я хочу в этот массив, значит, прочитать содержимое из файла. Здесь, соответственно, f read сам залочет, соответственно, повесит лог на файл, чтобы никто его больше не мог изменять в момент чтения. Значит, считает данные из этого файла столько, сколько я указал, и передаст их в файл, передаст их в мою структуру данных, в мой массив байтов.

Что важно здесь понимать? Файлы, исторически так получилось, что файлы, файловые системы, чтения вообще файлов, похоже на чтение с магнитных лент,

потому что это были первые носители у нас информации. Соответственно, там как было? Там была головка, которая просто двигалась вперед по магнитной ленте и читала данные с магнитной ленты, то есть один проход головки, считался одной единицей информации. Ну там это всего упрощенная модель, но тем не менее, плюс-минус это было так.

Соответственно, при чтении файлов, сейчас происходит то же самое. Вот у вас есть позиция в начале файла, она считается нулевой позицией, вы прочитали оттуда 7 байт, у вас позиция сдвинулась на 7 байт вправо, то есть как бы головка ваша воображаемая переместилась на 7 байт вперед. И следующее чтение файла произойдет уже с момента остановки предыдущей головки. То есть вот у меня есть test.txt, соответственно, вот у меня Mylineofcode. Первые 7 байт Mylineofcode – это раз, два, три, четыре, пять, шесть, семь, Mylineo.

Значит, я прочитаю 7. И здесь останется мой указатель, моя читающая головка, и вот это B2, который читается, он считается именно отсюда. То есть он прочитает значение FC.

Естественно, такие вещи мы можем перематывать. Ну то есть кассету можем перематывать, почему мы не можем здесь? Мы можем перемотать спокойно головку на любую позицию, можем перемотать на позицию, скажем, 6. Раз, два, три, четыре, пять, шесть, соответственно, ноль, один, два, три, четыре, пять, шесть, да, вот она, и соответственно, из нее тоже прочитать.

В чем разница между методом read и ReadAtLeast. ReadAtLeast попытается прочитать, как минимум то количество байт, который вы ожидаете, и вернёт вам ошибку если этого количества брать нету. Read может вернуть вполне спокойно пустой массив, если у вас уже был конец файла, если у вас, соответственно, ваша позиция, показывает на конец файла. ReadAtLeast, вернет вам ошибку.

Ну и соответственно, последнее, что хотелось бы рассмотреть. Соответственно, методом Seek можно перемотать вначало прекрасно. То есть метод Seek у нас работает, значит, следующим образом: он перематывает offset назад. И есть, так называемая, whence, в которой, передавая ноль вы указываете, что это вы считаете позицию этого offset-а от начала файла в один, вы читаете позицию от текущего offset-а, например, удобно, если у вас есть какой-то там последовательный файл. Например, CSV вы прочитали, а потом вы знаете, что у вас там есть бесполезные какие-то записи, вы точно знаете их количество, вы говорите: «Хочу от текущей позиции перемотать, вот это количество». Вам не нужно от начала файла что-то считать, складывать и так далее.

И два означает от конца файла, то есть там сдвинуться на минус сколько-то от конца файла, тоже так можно делать. Ну соответсвенно, перематываем здесь начало файла. И делаем, так называемый, Peek. Peek прочитает, значит, содержимое файла без перемотки, то есть, если у вас там в нуле, он прочитает 15 байт из нуля. И следующее ваше чтение будет снова начинаться с начала файла, ну и соответственно, после этого мы закрываем файл.

Давайте посмотрим, как это работает. Вот смотрите, как я и говорю, 7 байт MyLineo, потом мы читаем B2, у нас B2 это FC. Здесь можно вот эту вот штуку убрать, на самом деле, это не важно, slice у нас размером 2. Здесь, это не нужно.

Да, все правильно. Нет никаких новых переменных, соответственно, как-то так это будет происходить. Вот и видим, да, что потом я переместил на 6 позицию, переместился вот сюда. И прочитал of. И после 15 байтов соответственно, Mylineodcode и SO до сюда он прочитал. И следующий read будет читать снова сначала. В принципе, чтение Chunk-ами, так называемыми, это все, что вам нужно от файлов, от чтения файлов.

И давайте уже рассматривать запись в них.

Ну что ж, мы посмотрели, как читаются файлы. Давайте посмотрим, как пишутся файлы. Ну, собственно говоря, процедура абсолютно обратная. Да, у нас есть какая-то строка, какой-то объект. Ну как правило, это будет какая-то строка. Мы её переводим в массив файлов, вот таким вот образом кастуем. Строки кастуются напрямую, потому что строка и есть массив байтов, по сути. И пишем, пишем, соответственно, её файл.

То есть вот здесь как раз мы видим, что ставится разрешение, ставится разрешение абсолютно в Linux-овом стиле. То есть в данном случае владелец вам может читать, писать, остальные могут, соответственно, читать из файла.

И не на этом уровне у нас открывается файл с несколькими флагами. Здесь флаги, в принципе, достаточно олдскульные, то есть это такой массив байтов, которые определяют, что можно делать с файлом. То есть можно файл только писать, создать файл, либо, соответственно, когда он открывается…, не либо, а и, когда он открывается, мы его, соответственно, отрезаем, то есть мы его очищаем. И уже записываем заново все эти файлы, это всё атрибуты функции openfile. Функция openfile внутри себя уже содержит имплементацию. OpenfileNoLog – это как раз виндовая имплементация, здесь уже мы можем видеть различные, значит, обработки ошибок. И самое интересное это вот, например, у нас есть такие вот костыли для Windows. И вообще там много чего интересного.

В Windows есть ограничения по длине файла. Соответственно, оборачивается, permission в syscallmode, так называемые – стандартные разрешения для файла, виндовые. Ну и собственно говоря, на верхнем уровне у нас это всё, мы указываем путь к файлу, мы указываем, что туда записываем, permission и, собственно говоря, проверяем на ошибки.

Если мы сейчас запустим этот файл, мы увидим наш message, который записался. Если я сейчас туда что-то добавлю и перезапущу скрипт, как раз сработает этот (НРЗ 11:31). Все очистится. И у нас, соответственно, запишется файл.

Но опять же, если б всё так было просто, не было бы отдельной лекции. Давайте посмотрим теперь на более сложные применения записи файлов.

Ну что же, давайте посмотрим на запись. Соответственно, у нас есть файлик test, в нем что-то уже есть от моих предыдущих запусков. Предположим, что там ничего нету. Соответственно, создается файлик, то есть этот файлик у нас с помощью функции create, соответственно, создается. Если он уже есть, он просто обнуляется. С разрешениями 666, это значит, что все могут из него читать и писать. Соответственно, остается с флагами read, write, из него можно как читать, так туда и писать.

И, собственно, возвращается этот файлик. Мы опять же говорим, что в конце нашей функции его надо бы и закрыть. И первое, что мы делаем, создаём какой-то массив. И соответственно, пишем просто какие-то байты. То есть здесь точно также, как функцией read, мы пишем write. Мы пишем это файл. И тем самым, сдвигаем ту самую нашу головку, это работает не только на чтении, работает также и на записи, читающая головка, она же пишущая наши кассеты, магнитные ленты, все дела. Соответственно, пишется у нас массив файлов. Функция WriteString, соответственно, вам сама сконвертирует в массив байтов и вызовет метод Write. Такая оболочка позволяет записать строку. И метод f.Sync сольёт данные в хранилище. Почему так? Потому, что для более надежной работы с файлами в Linux-е, для более быстрой, вернее, работы с файлами в Linux-е и в Windows-е, существует специальная оболочка, которая держит файл в памяти.

То есть вы что-то редактируете, это не сразу записывается в файл, это держится в памяти операционной системы. И сливается память только в определенный момент, когда ядро вашей системы решает, либо когда вы непосредственно называете в Sync. То есть, когда вы говорите все, вот на этом моменте мне точно надо сохранить файл уже на диск. И тогда диск начинает шуршать, что-то там перезаписывать и себе его сохраняет. Соответственно, вот до этого момента вы не знаете, есть ли у вас там данные где-то в памяти, либо данные уже записались в файл на диске.

И вот эта функция вам точно говорит, что все, все сохранилось. Эта функция ничего не возвращает, возвращает только ошибку, если ошибки нет, соответственно, всё нормально. Здесь я ошибку игнорирую, присваивая безымянной переменой результат выполнения. И последнее, это у нас есть, соответственно, Reader-ы и Writer-ы, эти самые потоки. Иногда нам нужно записать поток, тогда мы создаём поток от файла. И что-то туда можем записать. Удобно, потому что потоки имеют свою презентацию в памяти, это не просто файл, вы можете эти потоки туда-сюда передавать, многие приложения с ними работают. И вызов методов вашего потока, соответственно, запишет всё в файлик. Давайте посмотрим, как это работает.

Соответственно, видим, первое, это some, writes, buffered, то есть у нас здесь записались все наши, соответственно, байты. Поскольку у нас здесь есть читающая головка, вернее, пишущая головка, мы точно также можем вызвать метод Seek для того, чтобы переместить позицию куда-то. И при таком при таком раскладе, значит, у нас последняя строка должна перезаписать то, что было перед ней. И вот давайте посмотрим, что получилось. Да, вот видите buffered перезаписала, перезаписала слово some. И, соответственно, оставила, какие-то там ошметки от второго нашего слова. Так что вот этим надо пользоваться, конечно, достаточно аккуратно.

В принципе, это всё, что я хотел сказать о записи файла. И увидимся на следующем занятии, в следующих блоках. Всем удачи и пока.

И хотелось бы пройтись, по тому, что мы сегодня узнали. Мы научились читать и писать файлы. Заодно посмотрели на разницу между Windows и Linux системами.

Соответственно, мы тем самым уже готовы к большинству задач OPS, потому что, как правило, задачи devops – это взять какой-то файлик, что-то с ним сделать и обратно записать данные. Большинство, ну естественно, не все. И на мой взгляд, этот блок очень важен. Я прошу ответственно подойти к выполнению домашнего задания для этого блока.

И увидимся в следующем блоке, всем удачи и пока.