Já entendemos os mecanismos básicos de como conseguimos ter concorrência e paralelismo. Mas na minha opinião o real problema é a coordenação. Não adianta nada você ter silício suficiente pra executar milhares de threads em paralelo se você não consegue coordenar o trabalho dessas threads. Então se tiver uma única coisa que você precisa levar dessa série toda é que a coisa mais importante nesse tipo de programação é conseguir a maneira mais simples e “barata” de coordenar tarefas concorrentes.

Vamos voltar ao que interessa. Você é um iniciante, finalmente entendeu os conceitos mais básicos de processos e threads. Aprendeu rapidamente sobre forks, sobre copy on write que ajuda a economizar memória. Mas você precisa entender alguns conceitos adicionais sobre coordenação agora.

A unidade mais simples numa CPU é uma instrução. Uma instrução em Assembly, que usa como argumentos certos valores que você preenche em determinados registradores. Daí a instrução executa e o resultado é escrito pela CPU em algum acumulador ou outro registrador. Ele incrementa o contador de programa e vai pra próxima instrução. É assim que a CPU começa a transformar o mundo ao seu redor. Tudo que você programa serve para pegar alguma informação que o mundo exterior vai fornecer, por exemplo o que o usuário digita ao programa, e você vai usar essa e outras informações para gerar algum resultado.

Programação é basicamente isso: transformação de dados. Uma instrução simples de soma, pega dois números que você armazena um em cada registrador, a instrução executa o cálculo e transforma esses dois números num único que representa a soma deles, e grava em outro registrador. Essa instrução é basicamente o que você chamaria de uma função. Toda função recebe parâmetros ou argumentos de entrada, processa alguma coisa e retorna algum valor de saída. E você pode usar o resultado de uma função como parâmetro ou argumento de outra função, e então vai encadeando uma função atrás da outra.

Quando você manda o sistema operacional executar o binário do seu programa, digamos um programa como wget ou curl no Linux, que você pode pensar como se fossem navegadores Web muito simples que rodam na linha de comando, você normalmente passa uma parâmetro, no caso uma URL de algum arquivo que você quer baixar. Você já viu ou vai ver um monte de tutoriais que usam um desses dois comandos pra baixar alguma coisa. O OS vai fazer o que já expliquei antes: alocar memória, criar um processo isolado, carregar seu binário, e passar esse parâmetro pro processo. Esse processo vai executar, no caso conectar no servidor que você mandou e baixar o tal arquivo. E quando terminar, o OS vai descarregar e liberar os recursos e o resultado, se tudo deu certo, pode ser um arquivo gravado em algum lugar no seu sistema de arquivos. Nesse caso o programa funciona como uma “função” também, ele tem uma entrada que é a URL que você passou de parâmetro e tem um retorno que foi o arquivo gravado. Ele transformou uma URL em um arquivo.

Em sistemas UNIX ou Linux, os shells como BASH ou ZSH, onde você roda na linha de comando, permitem encadear esses programas como se fossem funções. Quando seu programa carrega e vira um processo o sistema cria três canais unidirecionais: um canal padrão de entrada ou Standard In ou STDIN, um canal padrão de saída ou Standard Out ou STDOUT e também um outro pra erros, o STDERR. Na prática pense que todo processo pode iniciar conectando alguma coisa no STDIN e alguma outra coisa no STDOUT. Por exemplo, podemos usar o tal programa CURL pra baixar um arquivo da Web mas em vez de gravar em um arquivo local podemos redirecionar o conteúdo baixado pra esse STDOUT e conectar no STDIN de outro programa, por exemplo, um programa como o famoso grep que filtra texto pra fazer pesquisa. (curl -s https://somepage.com | grep whatever)

Quando ensinei sobre processos já expliquei que processos sobem em espaços isolados de memória e o código interno do processo não interfere no espaço de memória de outro processo. O que eu acabei de falar sobre encadear processos redirecionando o STDOUT de um ao STDIN do outro, usando o que chamamos de Pipes é a primeira forma mais simples de comunicação e, portanto, coordenação entre processos, mas não é o único e nem necessariamente o mais útil, porque você só liga o final de um programa ao começo de outro programa e esses canais só tem uma direção, ou seja, o primeiro programa é quem manda coisas pro segundo programa, mas não o oposto.

Sem entrar em muitos detalhes hoje, existem outras formas de um processo se comunicar com outro. Como já disse antes existem os sinais como SIGINT. E você pode codificar seu programa pra ignorar isso ou fazer alguma coisa quando recebe esses sinais ou eventos. Novamente é mais uma forma rudimentar de comunicação. Um terceira forma rudimentar é que dois processos abram um mesmo arquivo, daí um deles pode ir escrevendo uma linha nova de cada vez nesse arquivo e o outro processo pode ir lendo essas linhas novas.

Expandindo nessa idéia de um arquivo comum entre dois processos, o sistema operacional oferece outra opção, os FIFOs que se você estudou alguma coisa de estruturas de dados, funcionam como uma fila, First In, First Out, ou seja, o primeiro elemento que entra é o primeiro elemento que sai. Você pode abrir um FIFO e dar um nome e abrir o mesmo FIFO a partir de outro processo, é quase como se fosse um arquivo. Agora os processos podem se comunicar e a essa estrutura damos o nome de Named Pipe. E chamamos de pipe porque é como se fosse um cano mesmo, passando informação de um lado para o outro. Eles são limitados como os pipes simples ou arquivos. Um dos processos abre o named pipe pra ler e o outro pra escrever e por isso também a comunicação só flui numa única direção.

Em todos os exemplos, um dos processos serve de produtor de alguma informação e o outro processo serve de consumidor. E o mecanismo no meio, seja um arquivo ou um named pipe, serve meio como uma fila. Eu acho que o mecanismo mais simples de comunicação entre dois processos é uma fila e o sistema operacional já oferece esses mecanismos faz décadas. Porém, no nosso mundo de Web, internet e programas que funcionam como servidores - que seriam os produtores - e programas que funcionam como clientes - que seriam consumidores, esses mecanismos não funcionam, porque não se comunicam pela rede. Mas mesmo na mesma máquina, sem depender de rede, podemos simular a comunicação entre um servidor e múltiplos clientes usando named pipes ou arquivos. Porém, cada novo cliente conectando no servidor exige um novo named pipe ou arquivo. Por isso existem os sockets.

No caso da internet falamos sobre sockets de IP que trafegam pacotes no formato TCP. Uma das vantagens de usar sockets é que o programa servidor se liga a uma porta no sistema como já expliquei e aceita conexões de múltiplos clientes. Mas diferente de arquivos ou named pipes, o servidor só precisa se comunicar através de um único socket, a partir de onde vai aceitando as conexões de múltiplos clientes. Além de só precisar de um único socket pra abrir esse canal, a comunicação é bi-direcional, então o cliente também pode enviar dados pro servidor e vice-versa, pelo mesmo socket.

No caso de Unix e Linux além de um socket IP existe o que chamamos de Unix Sockets. Ele tem propriedades parecidas com sockets de IP mas só servem para processos rodando na mesma máquina já que não é um socket de rede. É a maneira preferida de fazer dois processos se comunicarem de forma eficiente porque você só precisa de um socket no processo servidor pra atender múltiplos outros processos clientes, que normalmente chamamos de workers, e ele abre um canal bi-direcional, diferente de named pipes. No episódio anterior eu expliquei como o servidor web NGINX tem um processo master, que serve como um servidor, e ele cria diversos processos workers e, claro, eles se comunicam entre si via esses Unix Sockets, canais versáteis bi-direcionais. É como outros servidores que geram forks e workers fazem para seus processos coordenarem entre si, como um Postgres ou MySQL.

No mundo web, onde hoje temos os tais micro-serviços, eles são basicamente processos, em programas servidores, que se ligam a uma porta de rede e abrem sockets IP aceitando conexões TCP e em particular oferecendo comunicação via o protocolo HTTP. Mas na prática são processos que se coordenam via sockets. Basta vocês entenderem que sockets, em particular sockets IP, obrigam um processo a se ligar a uma porta; e são canais bi-direcionais de comunicação. Em UNIX como BSD ou clones como Linux, podem ser Unix sockets para comunicação de processos na mesma máquina ou sockets IP para comunicação de múltiplas máquinas numa rede TCP/IP. E no nosso mundo Web essa comunicação normalmente se dá através do protocolo HTTP mas não é uma obrigação, podemos ter outros protocolos mais eficientes como o Protobuf ou Protocol Buffers que é um protocolo em formato binário, que é mais eficiente que o HTTP que é um protocolo texto, inventado pelo Google.

Aliás, lembrem que protocolos binários sempre são mais eficientes que protocolos em texto pelo simples fato que texto ocupa muito mais espaço. A explicação é muito simples. Num computador, representamos tudo no formato binário, 1 ou 0. Você já deve ter aprendido isso então vou resumir muito rapidamente. 1 byte são 8-bits. Em 1 byte você pode representar números inteiros de 0 até 255. Por que 256? Porque 2 elevado a 8 é 256. Uma palavra double byte ou seja 2 elevado a 16 representa números até 65 mil 536. Um número como 1,000, cabe perfeitamente em 2 bytes, se for representado binariamente. Mas, se você preferir mandar o número 1,000 como um texto, isso seria um texto de 4 caracteres. Em um sistema moderno que reconhece Unicode ou UTF-8, cada letra precisa de 2 bytes ou mais. Existe uma tabelona que mapeia cada caracter de cada alfabeto do mundo a um código. Em alfabetos ocidentais, 1 byte já resolve, mas kanjis de japonês precisa de pelo menos 2 bytes por exemplo. Mas no caso do texto 1,000 que é um, zero, zero, zero, são pelo menos 4 bytes. Em resumo, o número puro sozinho, em binário, cabe em 2 bytes, mas sua representação em texto precisa de 4 bytes, ou seja, o dobro do tamanho. Iniciantes tem dificuldade de entender isso mas comece decorando, o número inteiro 1000 é um elemento totalmente diferente do texto de 4 caracteres “1000”.

Isso fazia mais diferença numa época onde as redes eram extremamente lentas, e hoje você não pensa muito nisso porque qualquer um pode ligar nas porcarias de uma Net ou Vivo e pedir internet de 50 Megabits ou mais. Mas se você tem um sistema que trafega milhões de gigas por hora, todo byte faz diferença, por isso pra muita coisa o Google usa o Protobuf em vez de HTTP em seus sistemas. Pro seu dia a dia, não se preocupe tanto, HTTP tá de bom tamanho mesmo porque hoje sabemos comprimir texto com algoritmos como GZIP que podem derrubar o tamanho de um texto longo até menos de um décimo do seu tamanho. Mas isso é assunto pra outro dia.

Eu expliquei tudo isso porque a comunicação entre dois processos, usando um protocolo pesado como HTTP, sobre TCP/IP, passando pros sockets que seu programa se conecta, vai precisar fazer uma coisa chamada marshalling ou serialização. Ou seja, internamente no seu programa o dado, digamos o tal número 1000 puro, está numa variável que provavelmente é um inteiro de 2 bytes. Mas quando você vai enviar pra fora do seu processo, como a exigência é o protocolo HTTP, você vai serializar ou seja, transformar o inteiro binário pra um texto e cuspir pelo socket. Do outro lado um outro processo cliente vai receber esse texto e vai ter que converter de volta pra um inteiro, ou seja, desserializar o dado. Muitos dos bugs modernos em micro-serviços, ou seja, comunicação de processos através de HTTP, vem de erros nessa conversão e desconversão. Fique esperto com isso.

Mas eu estou me adiantando. Tudo isso foi pra explicar as diversas opções que existem para comunicação entre processos, você tem coisas muito simples como pipes, named pipes, arquivos, unix sockets e até sockets IP. Pra fazer um programinha besta, que só roda localmente, falar com outro programinha, que também roda só localmente na mesma máquina, obviamente usar um protocolo como HTTP é dar um tiro de canhão pra matar uma mosca. É descomunal e ineficiente. Na pior das hipóteses você vai querer usar Unix Sockets. Se você programa em C existem bibliotecas como OpenMPI e OpenMP para facilitar programar comunicação inter processos na mesma máquina ou mesmo entre diversas máquinas.

**16 min**

Entendido como processos se comunicam, vamos entrar dentro de um processo e falar de Threads. Eu já expliquei nos episódios anteriores que uma Thread tem acesso à toda a memória interna do processo. Então se uma thread precisa se comunicar com outra thread, elas podem basicamente compartilhar uma mesma estrutura de dados. No caso de C digamos um Array. Ou uma lista ligada. Como fica tudo no mesmo espaço de memória, não existe a necessidade de você usar mecanismos fora do processo como um named pipe ou unix sockets, porque você acessa o dado diretamente no endereço da memória virtual do processo. Mecanismos externos que eu falei antes existem porque um processo não enxerga nenhum endereço de memória externo ao seu espaço virtual alocado pelo sistema operacional. Por isso comunicação entre threads é ordens de grandeza mais eficiente, mas com essa eficiência vem todos os problemas de mutex, semáforos, bugs de race condition e dead-locks que eu já expliquei.

Programar com threads é um processo que tem que ser extremamente cuidadoso da parte do programador. O sistema operacional não tem como te proteger e você precisa garantir que uma thread não vai escrever em cima do mesmo endereço onde já tem outra thread escrevendo ao mesmo tempo. Num sistema que só opera com uma thread real, esse problema meio que não existe porque o scheduler do sistema operacional vai deixar só uma thread realmente rodar e a outra precisa esperar a vez, o tal do time-slice que eu já expliquei. Mas em máquinas com 2 cores ou mais, realmente 2 threads podem estar rodando exatamente no mesmo instante, com paralelismo real, e é aí que os bugs que eu falei podem acontecer.

Como eu disse antes, o problema todo é coordenação. Pra piorar, vocês vão notar que existe a CPU com seus cores reais, o hardware de silício, que rodam threads reais, uma por core. E existem as threads que o sistema operacional organiza e gerencia. O tal scheduler do sistema operacional como o CFS do Linux, tem acesso a esses cores físicos, mas os programas podem mandar criar 100 Linux threads. Então numa máquina Intel i5 da vida, com quad-core e hyper-threading, existem 8 threads reais que podem rodar em paralelo em cada determinado instante, mas o sistema operacional pode gerenciar centenas ao mesmo tempo, fazendo seu scheduler ir dando um pouco de tempo pra cada uma, fazendo o tal context-switching que eu expliquei, tirando as coisas da mesa de uma thread, recolocando as coisas na mesa pra outra thread e dando vez pra ela.

Para gerenciar essas threads, fazer o tal context switching, o sistema operacional precisa guardar o tal contexto em memória pra cada thread. E eu já disse que uma thread vai gastar pelo menos 1 MB de memória. Então quanto mais threads você criar mais memória vai gastar. Pior, quanto mais threads você criar, maior o trabalho de context switching. E pior ainda, quanto mais threads você criar maiores as chances de seu programa ter aqueles bugs por não ser thread-safe. Existe ainda outro agravante, criar threads é uma atribuição do sistema, mais especificamente da kernel. Para seu programa pedir pra kernel criar threads você precisa fazer system calls, as syscalls que eu falei, e isso também não é barato.

Aliás, se você não sabia, vale definir. Você pode pensar que a kernel do Linux, drivers, os programas que vem numa distribuição, e os próprios programas que você escreveu, é tudo a mesma coisa. São todos binários que rodam na máquina. Se você fez o que eu recomendei e estudou um pouco de Linux, você sabe que existe o comando sudo, o comando que escala seu programa pra rodar com permissões de root ou administrador da máquina. Nesse caso você sabe no mínimo que existem diferenças entre rodar como um usuário restrito normal ou rodar como um root que tem acesso a tudo.

Mas na verdade o root não é a coisa mais poderosa do sistema. O Kernel é que é. No final das contas é o kernel que decide o que o root pode ou não pode fazer. Então se você só chegou até o root, vamos entender outra coisa. A única coisa que tem acesso irrestrito a tudo da máquina, todos os dispositivos, toda a memória e pode executar qualquer coisa, é a kernel. Na arquitetura de processadores Intel existem 4 anéis de permissão. A Kernel boota e roda no chamado Ring 0, ou anel 0, dentro desse anel qualquer código tem acesso a tudo. Todos os seus programas, inclusive os que rodam via sudo, rodam no Anel 3, o mais restrito de todos. É onde rodam os seus programas. Nos Anéis 1 e 2 rodam os drivers e coisas de virtualização, por exemplo o Virtual Box carrega a kernel do OS que ele vai emular no Anel 1. No Anel 3 várias instruções de Assembly da máquina não funcionam, como a instrução que muda de anel, obviamente, ou a instrução HLT que dá Halt e pára a máquina. O sudo simplesmente libera mais system calls pra kernel, mas o programa ainda vai ter que fazer syscalls e pedir coisas pra kernel.

Num processador ARM também existem Rings mas a numeração é ao contrário, o que ele chama de EL0 é o equivalente ao user land, ou Ring 3 da Intel. O EL1 é onde roda a kernel, eu não tenho certeza mas acho que seria mais equivalente ao Ring 1 da Intel, porque no EL3 roda o Hypervisor que vai ser tema talvez do próximo episódio. Daí você pode rodar múltiplos kernels no EL1 isolados um do outro.

O mais importante é entender que existe User land que é onde rodam todos os seus programas e Kernel space, onde roda a Kernel, a única entidade que tem real acesso a tudo do sistema. Chamar uma função dentro do seu próprio processo é extremamente barato. Chamar uma syscall é mais ou menos como chamar uma função em outro processo, então você pode imaginar que vai acontecer algo parecido com um context switch toda vez. Fora isso, existe um salto em privilégios do Ring 3 pro Ring 0. Tudo isso custa. Ok, são micro-segundos a mais toda vez. Mas pense um servidor web com threads, com milhares de usuários pedindo requisições HTTP e seu servidor criando threads pra cada uma e matando todas que vão acabando. Milhares por segundo, milhares de syscalls por segundo, agora as coisas começam a ficar caras.

Não tem nada mais rápido do que código que roda dentro do seu processo. Por isso você quer evitar fazer syscalls o máximo que puder. Criar milhares de processos ou milhares de threads custa memória, custa context switch, custa escalar privilégios entre Rings. Parece que o povo do Node.js estava correto em usar I/O assíncrono, certo? Calma, I/O assíncrono e event loops ou reactors é só “Uma” forma rudimentar de concorrência em sistemas que não suportam outros mecanismos como threads. E justamente, Javascript não suporta threads. Ele é mono-thread, então a sua única opção é usar o intervalo entre atividades de I/O como arquivos ou sockets pra tentar fazer alguma coisa. Mas isso é muito pouco.

Eu já expliquei uma das soluções no episódio anterior. É fazer o que o NGINX faz: múltiplos processos, na média um pra cada core da sua máquina, ou seja, um processo pra cada thread e cada uma delas com um reactor pra aproveitar que chamadas de I/O costumam gastar tempo pra completar. A grande vantagem num sistema como Node.js que é um processo de um único-thread é que você tem alguma concorrência graças às syscalls de I/O assíncrono do sistema operacional, e pelo menos não tem que lidar com as dores de cabeça de mutex e bugs de sincronia como race conditions ou deadlocks. Imagine o inferno extra que ia ser se Javascript além de tudo ainda tivesse que lidar com mais esse tipo de categoria de bugs.

Interpretadores que dependem bastante do sistema operacional por baixo, como Python ou Ruby possuem Threads reais. Porém existe um grande problema. Para ter boa performance, eles delegam muita coisa pra módulos escritos em C. O Scipy, uma das bibliotecas científicas mais famosas do mundo Python é quase inteira escrita em Fortran e C++, por isso ela é veloz, só uma casca é escrita em Python. Essas extensões ganham acesso às estruturas de dados internas dos interpretadores, é como se fossem parte do interpretador.

Agora vem o grande problema: elas não são necessariamente thread-safe, na verdade a maioria não é, e você não tem como saber. Por causa disso, as Threads de Python e Ruby, apesar de mapearem para threads reais do sistema operacional e terem a capacidade de rodar em paralelo, acabam bloqueadas por um lock gigante do interpretador, que justamente ganha o nome de Global Interpreter Lock, ou GIL.

Com exceção de chamadas de I/O que conseguem rodar realmente em paralelo, as threads do interpretador são bloqueadas por causa do tal GIL. Elas ainda são úteis, mas não tanto quanto numa linguagem que não tenha o GIL. Na prática, assim como Javascript, tanto Python quanto Ruby acabam sendo essencialmente single-threaded, com algumas exceções como operações de I/O. Clones deles como Pypi ou JRuby, como não precisam manter compatibilidade com extensões em C, podem desligar o GIL e usufruir de threads reais.

Assim como Node.js em 2009, eu já expliquei como o Twisted em 2002 implementou a mesma solução de colocar um event loop numa thread pra atender milhares de requisições de I/O assíncronas e usar o intervalo enquanto espera essas chamadas completarem pra executar alguma computação e assim ter um jeito rudimentar de concorrência. Obviamente isso tem duas desvantagens, a primeira é que basta você fazer uma computação que demore demais e o event loop inteiro é bloqueado mesmo que a chamada de I/O já tenha terminado, já que roda tudo na mesma thread.

E a segunda desvantagem é que sua aplicação precisa ter muitas requisições de I/O pra compensar, ou seja, é melhor se usado em servidores de rede, como um servidor de aplicações Web. Esse modelo não funciona pra programas que são pesados em computação só. Por isso Node.js ou Twisted são indicados pra aplicações web. No mundo Ruby temos opções similares como Eventmachine ou o mais recente Async. No mundo Python, depois do Twisted também foi criado o Tornado e o Gevent que tem APIs mais modernas. Na prática é a mesma solução em todos eles: um event loop rudimentar que chamamos de reactor, que é gerenciado numa única thread, intercalando syscalls assíncronas de I/O e executando algum código nos intervalos.

Do ponto de vista da programação do código temos um problema. A forma como se programa um event loop é fazendo uma chamada de função assíncrona e passando como parâmetro uma segunda função que deve ser chamada quando o I/O acabar. Por exemplo, uma função de ler um arquivo de template de HTML na máquina recebe de parâmetro uma segunda função pra preencher esse template e gerar o HTML final. Num sistema single-thread, com tudo bloqueante, você escreveria essas chamadas de função uma embaixo da outra. Mas num sistema orientado a eventos, você precisa passar as funções subsequentes uma como parâmetro da outra. E isso cria o horroroso padrão de cascata ou o que eu gosto de chamar, de hadouken de funções. Se você ver algo assim hoje em dia, é um jeito antigo de escrever, é hediondo e não há ser vivo no planeta que veja algo nesse formato e não sinta imediatamente vontade de vomitar um pouco na boca.

No C já tínhamos algo parecido mas não necessariamente só pra concorrência, é o conceito de passar o ponteiro ou referência de uma função como parâmetro de outra função, chamamos isso de callbacks. Interfaces gráficas em VB ou Delphi ou Java Swing também tinham esse conceito com Listeners por exemplo. É um jeito rudimentar de se encadear funções onde uma depende do resultado da outra dentro de um loop de eventos. Toda interface gráfica é basicamente um loop também. Na prática, você tem um callback pra quando tudo dá certo e outro callback pra quando dá algum erro.

O Twisted do Python introduziu um conceito de código, os Deferreds. Em vez de chamar uma função, você cria um objeto que pode ser encadeado. Esse objeto basicamente engloba a tal chamada assíncrona, por exemplo, e você pode configurar esse objeto chamando métodos como addCallback. É quase a mesma coisa que antes, mas como estamos encapsulando a função num objeto, ele fica um pouco mais maleável de se trabalhar no seu código sem precisar virar um callback hell ou hadouken que falei antes. No mundo Web o framework JQuery adicionou a mesma construção com o mesmo nome, Deferreds. Assim você podia configurar uma chamada Ajax e conectar callbacks.

Na prática você encapsula uma chamada assíncrona num objeto, que você vai “deferir” a execução pra algum ponto no futuro, normalmente depois de ter a oportunidade de configurar esse objeto com **callbacks** ou esperar outros deferreds ficarem prontos e executar tudo de uma vez só, por exemplo. De qualquer forma é uma forma melhor do que ter hadouken de callbacks.

Retomando, fazer syscalls é caro. Context Switching é caro. Chamar coisas, funções, dentro do próprio processo é rápido e barato. Dentro do seu processo existe seu código transformado em binário. E você tem suas funções lá. Funções que demoram demais pra executar bloqueiam todo o seu processo ou thread. E se fosse possível pausar uma função antes dela acabar, deixar ela de lado, e dar chance de outra função rodar dentro do mesmo processo? Não estou falando de criar 2 threads, estou falando de na mesma thread a função ser pausada, deixar outra executar e daí voltar a executar a primeira. É como se fosse uma thread mas como é tudo dentro do mesmo processo, sem syscalls, sem context switching. É pra isso que serve uma corrotina, pense uma função com múltiplos pontos de suspensão, que você pode continuar ou resumir depois.

Em particular, um caso especial de corrotina se chama Fiber ou Generator. Se você já brincou com classes de Enumerators e Iterators em Python ou Ruby, sabe pra que elas servem. Elas são basicamente funções ou métodos como qualquer outro. Mas de dentro desse método faça de conta que existe um loop gigante, indo até infinito. Significa que esse método vai bloquear tudo e você não vai rodar mais nada pra sempre. Mas de dentro desse loop você pode chamar uma função, normalmente nomeada como yield, que vai devolver o controle pro código anterior que chamou o método. Agora esse código anterior pode executar outra coisa e quando quiser pode chamar o método resume, pra resumir a execução desse método com loop gigante, e ele vai dar yield de novo e assim por diante. Se você lembra de quando eu falei do Windows 3.1, isso seria algo parecido com uma multitarefa cooperativa. Fibers são funções, ou objetos de execução, onde você codifica pontos de suspensão ou pausa da execução. A maioria das linguagens atuais têm algo assim, os nomes vão variar mas se você ouvir corotina, fiber ou generator é basicamente a mesma coisa. Python tem, Ruby tem, Javascript tem, Swift tem, Kotlin tem, todo mundo tem algo assim.

De qualquer forma, pra que servem essas corotinas que pausam e podem ser resumidas se elas não rodam em paralelo? Em princípio elas se parecem com threads: elas estão dentro do processo, portanto tem acesso a todas as estruturas de dados do processo. Porém, como elas não rodam em paralelo, não existe a situação de uma querer escrever em cima dos dados da outra. Você que tem que manualmente pausar e manualmente resumir uma fiber, portanto você tem controle. Por isso elas são mais simples de se trabalhar do que threads. Mas elas não rodam em paralelo, então se parecem mais com as threads de Python ou Ruby, que apesar de terem a capacidade de rodar em paralelo já que mapeia pra threads reais do sistema operacional, existe o GIL que vai bloqueá-las. Mas diferente de threads, não precisamos de chamadas pra Kernel, elas são simples abstrações dentro do seu programa, em user-land.

A existência dessa construção na linguagem facilita a programação de concorrência. ?Eu não disse que a forma rudimentar é uma função recebendo callbacks pra chamar outras funções quando vier o evento que determinado I/O finalizou? Depois inventamos Deferreds que já ajuda. Porém, podemos usar Fibers pra escrever o mesmo código que seria um hadouken de callbacks quase como se fosse programação imperativa tradicional, só que o yield bloqueia até a chamada voltar e daí ele resume daquele ponto. Então veja no exemplo como é um código hadouken Agora veja o mesmo código usando fibers e a chamada de yield, e você vai ver como de repente do ponto de vista do código tudo ficou mais simples e sequencial. Então só por isso já temos um ganho absurdo em legibilidade e, por consequência, mantenabilidade do código.

Do ponto de vista do código, se somarmos um event loop com I/O assíncrono, conseguimos ter algum nível de concorrência e com fibers conseguimos ter um código que não é tosco. Estamos chegando em algum lugar aqui. Com isso Javascript com Node.js, Python com Twisted, Tornado ou GEvent, Ruby com Eventmachine ou Async, todos tem a possibilidade de usar essa arquitetura de reactors, porque no fundo quem realmente faz a mágica é a Kernel e suas syscalls assíncronas como epoll, kqueue ou IOCP. Porém, todas as bibliotecas que fazem chamadas síncronas, ou seja, do jeito bloqueante, precisam ser reescritas para usar esse recurso de fibers ou generators. Daí você entende porque em Python ou Ruby esse suporte é mais complicado e porque no Node.js é mais fácil: porque Node.js já nasceu com suas bibliotecas sendo criadas pra serem assíncronas.

Isso ajuda em linguagens que tem dificuldades com paralelismo de threads reais. Vamos falar das que tem acesso a threads reais sem o problema de um GIL ou lock global, como Java, ou C#. Pra elas existem Thread Pools. Em vez de ficar criando quinhentas threads a torto e a direito, sem controle, você limita quantas threads vão de fato existir simultaneamente. Como já disse antes uma média de uma thread por core da máquina. Tanto em Java quanto C#. Elas costumam ter alguma coisas como uma classe ThreadPool, literalmente um tanque de threads, e melhor ainda, quando tem outra abstração chamada Task ou Tarefa. A grande vantagem você vê num loop em uma lista gigante, onde você quer processar cada elemento dessa lista em paralelo. Só que se você ficar criando uma thread pra cada elemento, e a lista tiver quinhentos elementos, você vai acabar tento uns quinhentos threads pro sistema gerenciar. Ao passo que se você usar Tasks, que é só uma abstração, e configurar a ThreadPool pra ter só 5 threads, mesmo a lista tendo 500 elementos, você nunca vai ter mais que 5 threads rodando ao mesmo tempo.

Uma Thread Pool funciona como se fosse um load balancer de threads, vamos dizer. Nós usamos o conceito de Pool em tudo que é recurso caro. Por exemplo, bancos de dados nós usamos connection pools. Mesmo se tivermos 500 navegadores simultaneamente conectando no servidor de aplicação, não vamos criar 500 conexões no banco, vamos configurar um pool de conexões pra, digamos 100 conexões, e é só isso que vai existir, os primeiros 100 vão usar essas conexões, quando o primeiro acabar devolve a conexão pro pool e agora o próximo esperando vai pegar e usar e assim sucessivamente. A mesma coisa em ThreadPools. E esse é um dos principais conceitos que você tem que saber em arquitetura de concorrência e paralelismo: pools e filas. É como numa agência de banco. É dia de pagamento, 100 pessoas aparecem na hora do almoço, mas só vai ter 10 caixas. As 10 caixas representam o pool. E assim vamos tendo 10 pessoas sendo atendidas simultaneamente e as outras esperam na fila. É assim que o scheduler de threads funciona também. E é assim que podemos ter algum controle sobre as threads do sistema operacional.

No mundo .NET você tem classes como Parallel e Task que abstraem esses conceitos. Em Ruby temos também uma biblioteca chamada Parallel que vai abstrair pools de forks ou pools de threads. No mundo de Mac e BSD existe o Grand Central Dispatch e seus DispatchQueue. Enfim, cada linguagem ou sistema operacional vai oferecer alguma coisa que tem essa arquitetura. Alguma combinação de queues ou filas e pools. Preste atenção: você raramente quer usar Threads reais diretamente, especialmente em grandes quantidades. Se precisar de threads sempre use Pools, Queues e Tasks.

Esse conceito também se estende pra computação distribuída. Até agora sempre falamos de um código que, na hora que precisa, carrega alguma forma de executar uma tarefa em paralelo. Ou um fork pra gerar um novo processo. Ou criando uma nova Thread. Ou agora criando Pools. Mas existe a opção de já ter essa segunda entidade pré-carregada e esperando. Um segundo processo, ou uma thread ou algo assim.

Em Java sempre tivemos no Enterprise Edition os JMS ou Java Message Service. No mundo Ruby on Rails usamos coisas como Sidekick, a aplicação conecta a uma fila representada num banco de dados Redis e existe um ou mais processos que ficam pré-carregados e ouvindo nessa mesma fila. Chamamos esses processos secundários de Workers. Temos isso em quase toda linguagem. Uma fila e vários workers, e como é um número limitado de workers é como se fosse um Pool também.

No mundo .NET temos opções como o projeto Hangfire. No mundo Python temos projetos como Celery ou RQ. No mundo Node.js temos coisas como o Kue. No mundo Java temos Message Queues no JEE ou Background Jobs no Spring. O conceito é sempre o mesmo: algum tipo de fila como RabbitMQ, Redis ou qualquer coisa assim, e diversos workers consumindo dessas filas. É uma solução de paralelismo que funciona tanto na mesma máquina quanto em diferentes máquinas, representando um tipo de computação distribuída. Outro nome que você vai ouvir associado a esses mesmos conceitos é Background Jobs.

Esse tipo de solução resolve a grande maioria dos problemas reais que temos hoje. Mesmo numa aplicação mono-thread, que tenha um reactor, ainda temos operações que custam tempo e processamento, que eu já expliquei que vão bloquear o event loop. Em vez de executar isso no processo principal, podemos enviar uma mensagem a uma fila e deixar outro processo, ou worker, fazer esse trabalho pesado, e assim deixar o reactor principal mais leve e mais rápido.

Mas ainda assim gostaríamos de alguma coisa mais eficiente dentro de um mesmo processo. E até agora não chegamos lá ainda. Mas se você prestou atenção aos episódios anteriores, existe sim uma construção pra definir concorrência em sistemas operacionais e hardware que não tem mais do que uma thread real. Uma green thread. Ou seja, uma thread que só existe em User land e não em kernel space. É o que o protótipo do Java original tinha no começo quando foi desenhado pra set-top-boxes, tipo Tivo como eu expliquei em outro episódio. Um conceito de hardware barato do começo dos anos 90 que não rodaria nada muito mais potente que um DOS. Se quiséssemos programar concorrência, não haveria threads reais, mas mesmo assim podemos programar concorrência com green threads, só que elas nunca vão rodar em paralelo. Lembram? Uma coisa é ter concorrência, outra coisa é ter paralelismo. Ter paralismo implica ter concorrência, mas não o oposto.

Corotinas, fibers ou generators que expliquei acima, seriam mecanismos para chegar em green-threads: funções com pontos de suspensão, onde podemos gerenciar várias delas ao mesmo tempo, e ir executando um pouco de cada uma de cada vez. À primeira vista, hoje que temos máquinas multi-core com múltiplas threads paralelas reais, pra que serviriam green-threads? Pra muita coisa. Eu já repeti várias vezes e vou repetir de novo: threads custam memória, talvez 1 MB, tem context switching caro, executado pelo scheduler da kernel, tem syscalls que precisam pular de user-land pra kernel space. Green threads são baratas, dependendo da linguagem pode custar talvez na ordem de 2 quilobytes, ou seja umas 500 vezes mais leve que uma thread. Elas estão dentro do mesmo processo, em user land, e não dependem de syscalls nem de context switching da kernel. Ou seja, comparado com threads, elas são realmente muito mais baratas. Mas tem um problema: só com o que sabemos até agora, elas não rodam em paralelo.

Aí que você se engana. E se, usássemos o conceito de thread pools e fizéssemos um código que pegasse milhares de green-threads e mandássemos executar num pool de threads reais? Basta construirmos um scheduler em user-land que se encarregaria de fazer esse load balancing de green-threads entre threads num pool. E muito bem chegamos à base de um Scala, Clojure, Go ou Erlang e, por consequencia, Elixir.

Erlang, nos anos 80, fez o seguinte: quando sua VM Beam sobe, ele sequestra os recursos da máquina pra ele: o I/O, a memória e principalmente as threads reais do sistema operacional. Ele roda um scheduler em user land pra cada thread real. As funções em Erlang não são só fibers, elas são corotinas, e a VM tem o poder de arbitrariamente pausar uma função quando bem entender, ou seja, ela não depende que você tenha programado um yield na função pra devolver o controle, ela vai pausar quer você queira ou não, exatamente como a Kernel do Linux faz com seus programas. Ou seja, o Erlang implementa o equivalente a um scheduler preemptivo de green threads em user land, e em vez de fibers ela tem corotinas de verdade.

Mas o Erlang tem algo mais importante ainda. Um dos problemas de uma thread é que se ela tem problemas, exceções, erros e outros bugs, ela pode corromper toda a memória do processo. Então basta uma thread fazer bobagem e seu processo inteiro pode crashear. Pra evitar isso o Erlang resolveu que cada uma dessas suas corotinas não compartilhasse nenhuma memória com outras corotinas. Na verdade, pra dificultar nosso vocabulário, dentro do Erlang ele chama essas corotinas que estão executando de “processos”, não confunda com processos de verdade de um Linux, apesar do comportamento ser similar. Existem funções normais como de qualquer outra linguagem, mas existem esses processos de erlang. Como esses processos não enxergam memória de outros processos, se você quiser fazer um processo se comunicar com outro temos o equivalente a Unix Sockets se fossem processos de Linux, canais bi-direcionais por onde eu posso enviar mensagens.

Essas mensagens são estruturas de dados, de novo, não compartilhadas. A VM do Erlang, assim como um OS Linux, aloca espaços de memória protegidos pra cada processo. Mas diferente do kernel Linux, que aloca 1 MB pras estruturas de um processo ou thread, o Erlang não precisa alocar mais que 2 quilobytes pra cada processo. O processo vai recebendo mensagens num mailbox, um tipo de fila e escolhe o que fazer com essas mensagens quando quiser. Quando um processo termina ou crasheia, existe um garbage collector pra cada processo individualmente. E isso pra mim é o que diferencia Erlang de todo o resto, um processo, que é um green-thread, com bug não afeta e nunca corrompe o resto do sistema já que ele não compartilha nada.

Eles não compartilham ponteiros nem nada e por isso nunca tem a capacidade de desestabilizar o sistema. Por isso eu costumo dizer que a VM Beam do Erlang é quase como se fosse um mini-Linux. Pra ficar mais similar ainda, os Linux tem um processo master que dependendo da distro pode ser um initd ou systemd ou outro que sobe daemons, que já expliquei antes, são processos especiais, serviços do sistema. No Erlang temos a mesma coisas só que se chamam Supervisors. Eles se encarregam de quando você subir sua aplicação, no boot da VM do Erlang, vai subir os supervisors que vão iniciar os processos necessários e garantir que se um deles crashear ele vai subir um novo processo no lugar.

Eu expliquei em 2 minutos o Erlang, mas pense em Erlang mais como um Java: uma máquina virtual que te oferece diversos serviços, incluindo coisas que o Java ou .NET não tem: schedulers e processos. Esses processos são green-threads. Tudo roda em user-land, e por isso é extremamente barato. É tão barato de fato que podemos fazer o que eu expliquei que fazíamos antigamente: toda nova conexão de rede antes a gente gerava uma thread real pra lidar com ela. Mas com o C10K descobrimos como é caro e não escala ficar criando threads reais o tempo todo pra toda conexão. 10 mil threads consumiria 10 gigabytes de RAM. Mas 10 mil processos Erlang consome só 19 Megabytes. Portanto, como essa green-thread é ordens de grandeza mais barato, podemos voltar a criar uma green-thread por conexão se quisermos. Entenderam a diferença? Green-threads são ordens de grandeza mais eficientes que uma thread real, e se a máquina virtual tem seu próprio scheduler pra gerenciar a thread-real pra rodar as green-threads temos paralelismo de verdade.

O Go faz mais ou menos a mesma coisa. O que o Erlang chama de processo, em Go se chama goroutine. Só que eu diria que as goroutines são mais baixo nível do que um Actor de Scala. Num Erlang, assim como num Linux, cada processo tem um PID ou Process ID, um número que identifica unicamente cada processo rodando. Esse PID serve como um e-mail, e um processo pode mandar mensagens pra outro processo através desse PID. O Erlang é tão versátil que podemos mandar mensagens pra um PID de um processo em outra máquina.

O Go tem uma outra abstração: as channels, literalmente canais. Duas gorotinas podem compartilhar uma mesma channel. Uma gorotina escreve num channel e outra gotina pode ficar escutando do mesmo channel. Lembram de algo similar que expliquei hoje? Na prática não me parece nada muito diferente de um named pipe entre processos Unix, um canal unidirecional bloqueante. E eu disse que gorotinas e channels são mais baixo nível que Actors de Erlang ou Scala porque eles permitem trafegar qualquer coisa, incluindo ponteiros de memória, ou seja, eles compartilham memória entre si e, portanto, tem os mesmos problemas que programar com threads reais: se você escolher trafegar ponteiros, vai precisar lidar com mutexes e locks e os temidos bugs de race condition e deadlocks.

Diferente de Erlang ou Java, o Go não é uma máquina virtual, cheia de serviços. A filosofia e os usos são diferentes. Ele tem um runtime por causa do controle das gorotinas que podem rodar em paralelo. Diferente do scheduler de Erlang que é preemptivo, o scheduler de Go é cooperativo. Ele espera determinados eventos numa gorotina como syscalls pra receber o controle de volta e dar a vez pra outra gorotina. É menos sofisticado do que Erlang nesse caso mas é mais leve e mais previsível também. E obviamente, por causa disso ele é automaticamente mais poderoso do que Node.js e mais simples de programar aplicações complexas. Assim como Erlang ou Scala, um reactor é desnecessário, porque eles tem schedulers que podem pausar uma corotina quando alguma coisa como I/O bloqueante é executado. Na minha opinião, schedulers em user-land pra corotinas são melhores do que reactors. Reactors são interessantes em linguagens que não tem como ter schedulers em user-land. A premissa pra isso é acesso irrestrito às threads reais, sem locks globais que Python ou Ruby tem.

Do ponto de vista puramente dos interpretadores ou máquinas virtuais. Temos PHP, Python, Ruby, Perl e Javascript que são todos interpretadores. Alguns tem acesso a threads reais, mas na prática, todos são essencialmente single-threaded, com locks globais que chamamos de GILs por conta das antigas extensões em C. Em todos eles, a opção de um Reactor com a ajuda de Fibers pra não complicar a programação, é a melhor solução pra concorrência. Erlang e Java são máquinas virtuais, no caso de Java significa que Scala, Clojure ou mesmo Kotlin, todos rodam sobre a mesma máquina virtual. Essas VMs são excepcionais trabalhos de engenharia e foram feitas pra sequestrar todos os recursos da máquina. O melhor caso de uso é seu programa e a VM rodarem sozinhas na máquina e elas gerenciarem tudo. E mais do que isso: o melhor caso de uso é que uma vez iniciadas elas não fiquem rebootando toda hora. Guardem essa informação pro episódio de devops.

Erlang, na minha opinião, é quem tem as melhores abstrações de concorrência, e por consequência Elixir também já que ele basicamente gera bytecodes pra mesma VM, usando os mesmos frameworks. Em Java, Scala e depois Clojure e Kotlin adicionaram abstrações de linguagem e bibliotecas pra facilitar concorrência, inclusive no caso de Scala e Clojure implementando schedulers user-land e passando a adotar o conceito de green-threads, que o Java nativo não tem. O C# veio copiando cada uma dessas coisas de outras linguagens como Fibers, ThreadPools e tudo mais, mas que eu me lembre ele também não tem schedulers em user-land nem green-threads. Kotlin, Swift, C# tem sim classes chamadas schedulers que não é a mesma coisa que eu falei de Scala ou Go. Eles não são serviços do runtime, eles são classes pra agendar coisas pro futuro, como um cron num Linux, onde vc diz pra determinada tarefa rodar numa thread num determinado horário ou numa determinada frequência. Os nomes confundem, eu sei, mas schedulers de Kotlin, por exemplo, é como se fosse um cron.

O Go é uma linguagem que compila binário nativo, assim como Rust ou C. Mas o Go possui um runtime, pense um pedaço grande de código que gruda no seu pra gerar o binariozão final. Ele tem um scheduler real cooperativo e oferece green-threads com coordenação via channels, que como eu disse, parece com named pipes como eu expliquei no começo do episódio. Rust e outras linguagens como Crystal ainda Não tem o mesmo nível de concorrência e paralelismo. Mesmo em Rust o suporte a I/O assíncrono demorou pra chegar. Crystal está agora ganhando suporte experimental a rodar fibers em paralelo, mas nenhum dos dois possui schedulers em user-land que vem pré-embutido no runtime da linguagem como GO.

Se não ficou claro, Javascript não tem paralelismo. Ele é a mais simples de todas as linguagens, e muito do investimento em engenharia no Google se focou em tornar a V8 muito boa em cuspir código nativo, portanto ele é um Just In Time Compiler, um JIT muito bom. Ele não tem sintaxe complicada, não tem bibliotecas padrão pra manter, e obviamente não oferece scheduler user-land também. Não tem um GIL como Python ou Ruby, mas também não tem acesso a threads reais. Sua única forma de concorrência são generators ou fibers, em multitarefa cooperativa, e na falta de um scheduler, essa multitarefa é controlada num event loop reactor. No máximo dá pra fazer fork pra outros processos. Porém, Javascript troca performance por uso de memória, então Copy on Write do Linux Não ajuda muito porque ele faz JIT, então muita coisa não pode ser reaproveitada entre processos. Por isso eu tendo a dizer que apesar de ser rápido, é uma das linguagens mais rudimentares que temos.

Resumo

Eu ia falar sobre a parte da programação, mas como já tá comprido pra caramba de novo vou só resumir. Eu expliquei até agora sobre como as coisas funcionam por baixo. Agora como tudo isso fica no seu código? Se lidarmos com threads reais, já expliquei que precisamos ficar programando mutexes pra lá e pra cá. Em linguagens antigas como Java ou C#, que expõe Threads reais, você ainda precisa se preocupar com isso. Em Go também você precisa se preocupar com isso, porque ele transporta ponteiros nos channels entre gorotinas. Ou seja, o oposto de um Erlang ou Elixir que não compartilha nada entre seus processos. Portanto ficou na sua mão como desenvolvedor se virar com sincronização e coordenação. Experimente criar um deadlock no seu channel de Go: seu programa crasheia. É um dos motivos de porque eu fico decepcionado com Go em 2019 que já poderia ter oferecido abstrações melhores direto na linguagem. Mas é uma opinião pessoal, o mesmo problema acontece num C ou C++ onde eu tenho acesso a tudo, incluindo ponteiros. Esconder ponteiros é uma decisão de design da linguagem. Mas assim como GOTO desapareceu, Null é considerado ruim hoje em dia, Ponteiros também já haviam sido aposentados quando Java foi criado, não é nenhuma novidade.

De qualquer forma, Node.js no começo, por 2010, tornou famoso o estilo porco de programação com callbacks. Twisted e JQuery tornaram famosos a abstração de Deffereds. E Scala, não lembro se foi o primeiro, mas foi um dos que ajudou a popularizar o termo Future. Não deixa de ser uma pequena evolução sobre Deferreds. Na prática é mais uma forma de encapsular uma execução futura num objeto, um tipo de placeholder. Em vez de fazer sua variável receber direto o resultado de uma chamada assíncrona ou mesmo uma chamada externa tipo uma chamada HTTP, que não se tem certeza se vai completar, ou uma query num banco, você faz a variavel receber um Future e o Future encapsula essa chamada incerta. Quando o Future executar e resolver, ele é substituído pelo resultado.

Ninguém mais usa os nomes Deferred ou Futures ou mesmo Delay, o próximo passo nessa evolução são Promises, em particular o famoso spec A+. Se você está em Javascript faz alguns anos, já deve estar de saco cheio de ouvir falar de Promises. Dois ou três anos atrás eu dizia que a ironia de Promises é que eles permaneciam sendo promessas. Enfim, na prática, Promise é como eu já falei antes de Deferred e Futures, um objeto que encapsula alguma chamada incerta. Esse objeto é encadeável e customizável e costuma ter métodos como then e catch, que é onde você adiciona callbacks, como o addCallback do Deferred de Twisted. Java 8 tem CompletableFuture, frameworks como Guava tem SettableFuture. Javascript tem Promise/A+.

Deffereds, Delays, Futures, Promises, são termos que acabam sendo usados um no lugar do outro, mas hoje em dia as pessoas falam mais de Promises. Sobre os objetos de Promises podemos adicionar o que chamamos de syntactic sugar, ou literalmente um açúcar na sintaxe da linguagem pra não ter que lidar com objetos de Promise diretamente nem essas configurações via then ou catch. Se não me engano a Microsoft desenvolveu a sintaxe de async/await pro C# e várias outras linguagens implementaram algo similar.

Voltando um pouco, em linguagens que tem Threads costuma também existir a função de join. Por exemplo, você pode criar um array ou lista com 10 Threads. Ao criar todas elas, o código pai que está criando pode chamar Join em todas elas. Assim o programa principal vai ficar bloqueado até todas elas executarem e retornarem. Daí ele continua a execução com o resultado de todas as threads, que garantidamente já acabaram de rodar. Async e Await é quase a mesma coisa, funções marcadas como async costumam devolver um objeto Promise e o await é como se fosse um join numa thread, ou um yield num Fiber, é tipo um join numa promise, esperando que ela resolva pra depois continuar.

A diferença é que async/await é uma semântica que pode ser usada em threads, green-threads ou corrotinas ou fibers. Então é como se fosse Join pra coisas que não são threads. Todo programa que você escreve são trechos de código que chamam outros trechos de código, uma função chamando outra. Pra explicar um exemplo, vamos chamar a primeira função de Maria e a segunda função de Joao. Maria chama Joao, Joao executa, e devolve controle pra Maria. Se Joao for uma Fiber, Maria chama Joao, Joao começa a executar e dá yield, suspende e devolve controle pra Maria. Agora Maria pode fazer outra coisa e, se quiser, pedir pro Joao resumir de onde parou.

Com Promises é assim: Maria chama Joao, Joao não faz nada e devolve uma promessa. Maria pode escolher quando cobrar a promessa. Daí ela cobra, e Joao executa o prometido e quando termina “resolve” devolvendo o resultado. Agora Maria pode escolher o que fazer com o resultado. Ela tem duas opções, ou usar a sintaxe padrão de Promises e dizer ao Joao: Joao quando você terminar chama a Camila. Ou, a Maria pode cobrar a promessa do Joao, esperar ele terminar que é o await, e quando terminar ela mesma pode chamar a Camila. Entenderam, é uma questão de estilo de código, ambos vão chegar no mesmo resultado, mas hoje em dia se tornou consenso que se precisar fazer uma chamada externa, por exemplo uma requisição de rede, que não sabemos quando vai ficar pronto ou mesmo se vai completar, então englobamos num Promise. Daí escolhemos se esperamos já ele terminar ou se só atrelamos nele um callback e vamos fazer outra coisa.

Processos, threads, user-land e kernel-space, context switching, schedulers de user-land, corrotinas, fibers, generators, são construções do sistema operacional, interpretador, runtime ou máquina virtual pra modelar e executar concorrência e paralelismo. Recursos das linguagens como filas, mailboxes e channels, promises e async/await são recursos das linguagens pra facilitar a programação de concorrência e paralelismo.

Finalmente, chegamos ao final. Espero que tenham aprendido o suficiente pra começar a entender como as diferentes linguagens lidam com concorrência e paralelismo, o que de fato acontece por baixo dos panos, e não misturem o que uma linguagem pode ou não pode fazer. Na prática, hoje em dia, todas as linguagens tem recursos muito bons. Alguns servem pra coisas que os outros não são tão bons, então nenhuma das linguagens que falei aqui é obrigatoriamente melhor do que a outra. Não adianta ser só rápido. Não adianta só ter concorrência massiva. Uma boa linguagem, primeiro, precisa de um ecossistema que gera resultados com eficiência. Quanto menos desperdício, quanto mais resultados, pra determinado caso de uso, melhor.

Estamos já faz muito tempo nesse assunto de back-end e o assunto de concorrência se esticou bastante. Ainda estou em dúvida se semana que vem ainda adiciono mais um tema, que é relacionado a gerenciamento de memória, ou se já vou direto pro assunto final da série: devops. Vou pensar ainda. Se ficou com dúvidas não deixe de mandar nos comentários abaixo, se curtiu mande um joinha, compartilhe com seus amigos, não deixe de assinar o canal e cilcar no sininho pra não perder os próximos episódios. A gente se vê semana que vem, até mais.