Recapitulando, eu já expliquei várias vezes como seu código fonte compila num binário e esse binário é carregado em memória. Também já rapidamente expliquei como CPUs e sistemas operacionais modernos alocam espaços de memória isolados pra cada processo, com endereços virtuais de tal forma que um processo não tem como acessar a memória de outro processo. Como eu já disse antes, se você realmente pretende levar a sério a profissão, precisa entender como seu computador realmente funciona.

Uma coisa que eu nunca mencionei por exemplo é o processo de boot em si, que é bem fascinante. Em hiper resumo, tem diversas etapas, do momento que você aperta o botão de ligar, segue o processo de Power on self test ou POST que você vai ouvir falar bastante se acompanha canais de montar seu computador, tem o teste de memória e outros dispositivos, até chegar no Master Boot Record que é o antigo MBR ou o atual GUID Partition Table ou GPT pra carregar o boot loader, que no caso do Linux seria um GRUB2 por exemplo, e finalmente carregar o binário do Kernel propriamente dito que, por sua vez, vai carregar um systemd, que por sua vez finalmente vai carregar todos os daemons pra terminar de tornar disponível os recursos da máquina pros programas, como montar suas partições do disco. Se você está estudando Linux, veja os links que deixei na descrição abaixo que detalham esse processo em mais detalhes.

Eu estava pensando em quanto de detalhes quero mostrar hoje, mas se eu for realmente minucioso o episódio vai ficar com horas e horas de duração e vai ser extremamente tedioso. Em vez disso vou delinear os pontos principais e explicar resumidamente, você precisa pesquisar mais detalhes depois nos seus estudos.

Em termos de memória, entenda que você tem os pentes de RAM, o que você normalmente vê como pente de 16GB DDR4, DDR que significa Double Data Rate. De curiosidade eu já falei rapidamente antes que tudo no computador funciona segundo o clock da CPU. Sabe quando a gente fala em CPU de 4 Ghz? Sendo hertz um ciclo, 4 Ghz significa 4 bilhões de ciclos por segundo, pense em clock literalmente como um ciclo ou uma ida e volta de um pendulo, uma frequencia. Double Data Rate significa que é possível puxar dados da memória no topo do ciclo e na parte de baixo do ciclo, em vez de só conseguir puxar dados uma vez por ciclo como era antigamente, então ele tem a chance de puxar duas vezes, por isso é mais rápido, mas hoje em dia a gente só usa DDR de qualquer forma. Uma diferença pra memórias de servidor são as memórias ECC que significa Error Correcting Code. É uma memória bem mais cara mas que garante que a memória nunca vai ser corrompida, pentes de RAM normais como as que usamos tem chances de devolver 1 bit errado de tempos em tempos, mas é raro, memória ECC de servidor tem uma proteção extra e menos chances.

O que o seu simples programa vê não é só RAM, é mais um “espaço de memória”, a CPU vai cuidar do que vai aonde. Você tem caches L1, L2 e L3 que são memórias pequenas e hiper rápidas que ficam no mesmo chip, pra fora dele você tem um barramento, que vai conectar com dispositivos como os pentes de RAM e com seu HD mecânico ou SSD que também pode servir de memória se conter um arquivo de SWAP. Se você não sabia disso memória é dividida normalmente entre acesso rápido e devagar. L1, L2 e L3 são memórias ultra rápidas. Quanto mais rápido mais caro, lógico, por isso eles são muito pequenos da ordem de até 64 quilobytes de L1, até meio megabyte de L2 e até 8 megabytes de L3. Eles não são memórias pra você usar, o CPU vai usar principalmente como cache pra manter seus pipelines cheios. Depois disso vem a RAM propriamente dita, seus pentes de DDR4, ordens de grandeza mais lentos que os caches, mas de uso geral. E depois vem coisas como o SWAP do seu SSD que é ordens de grandeza mais lento que a RAM.

Eu gostaria muito de explicar em detalhes o que são portas lógicas, como se chega num flip-flop, e como flip-flops podem ser usados pra armazenar bits. Pra ter um modelo na cabeça, pense em memória como um livro, com um espaço reservado pra um índice, e cada linha apontando pra uma página desse livro. Essas linhas são as linhas de endereço. Vamos entender essas linhas de endereço e como elas são organizadas.

Quando falamos que uma CPU é de 64-bits significa que os tais registradores da CPU acomodam números de até 64-bits de tamanho, e além disso temos barramentos, ou seja as rodovias de comunicação entre a CPU e a memória e dispositivos de I/O que também tem um tamanho máximo. 64-bits significa que a CPU consegue lidar com words ou palavras de 64-bits inteiros, que equivale a 2 elevado a 64 que seria o absurdo número de 16 Exabytes. Pense assim, você tem bytes, quilobytes, megabytes, gigabytes, terabytes, petabytes e exabytes. 16 Exabytes é este numerazão aqui embaixo se for em bytes (18.446.744.073.709.551.616).

Agora, esse numerozão está representado no sistema decimal, que é o que estamos mais acostumados a lidar, mas quando falamos em computador, ele representa tudo em 1 ou 0. Vamos escovar bits um pouco. O número 1 é só 1, o número 2 é 1-0. O número 11 é 1-0-1-1. Pense em binário como a tabuada do 2, o número mais pra direita é 1, o segundo da direita pra esquerda é 2, na sequência vem 4 e depois 8 então 1011 é 8 + 0 + 2 + 1 que é 11, entenderam? Portanto o menor número de 64 bits é basicamente 64 zeros em sequência e o maior número seria 64 1s em sequência.

Obviamente ficar representando tudo em binário no papel é muito longo. Como lidamos com bytes que são conjuntos de 8 bits, é mais fácil lidar com uma representação que é múltipla de 8 ou, melhor ainda, múltipla de 16, que é o que chamamos de hexadecimal. É um sistema que em vez do nosso decimal que vai só até 10 ele vai até 16, mas como números arábicos só tem caracteres de 0 a 9, precisamos usar as letras do alfabeto de A até F pra ir até 16. O mesmo número máximo de 64 bits que eu falei que é 64 vezes o caracter “1”, em hexa seria quatro grupos de quatro Fs.

Tão vendo? Aquele numerozão horroroso em decimal que você jamais vai decorar é muito mais simples em hexadecimal e você consegue literalmente saber de cabeça que vai de 0 até 16 Fs. É muito mais curto. E em binário você tem propriedades matemáticas interessantes, por exemplo, multiplicar um número binário por 2, é simplesmente um shift left, tipo mover tudo uma casa pra esquerda e colocar um zero. Por exemplo o número 11 que eu falei que é 1-0-1-1 vira 1-0-1-1-0 que é 16 + 0 + 4 + 2 + 0 que é 22. Algumas coisas ficam mais simples de calcular.

Você precisa entender que usamos o sistema decimal que vai de 1 até 10 por motivos históricos. Vale pesquisar a história da matemática, você vai encontrar outros sistemas. Um exemplo, se você usar o relógio como parâmetro, é mais fácil contar de 1 até 60. Computadores são 1 ou 0 porque é o sistema mecânico mais simples, pense o interruptor de luz da sua casa, em todo instante ele armazena uma de duas informações: ligado ou desligado. 10 interruptores guardam 10 bits de informação. E os computadores mais rudimentares do passado usavam lâmpadas a vácuo, como o que ilumina seu quarto neste instante. Luz ligada ou desligada. Coloque 10 lâmpadas e você pode visualizar 10 bits de informação.

Continue estudando mais sobre representação binária e hexadecimal, é importante você ter essas ordens de grandeza na cabeça de forma natural, mesmo que seu trabalho não seja fazer aritmética direto nesses sistemas. De qualquer forma, estávamos falando sobre computadores 64-bits. Este número abaixo (16 Fs) é o maior número inteiro que é possível armazenar em uma palavra de 64-bits e voltando às linhas do índice do nosso livro, significa que cada linha de endereço pode apontar números só até esse tamanho e, por consequência, esse tamanhão máximo de RAM ou páginas do nosso livro. Só que isso é teórico.

Nenhuma máquina que eu saiba tem capacidade pra realmente comportar esse tanto de memória real. Um dos motivos é porque embora o processador tenha registradores de 64-bits, o barramento entre a CPU e a memória em processadores Intel, se não me engano, é de 42-bits, e na AMD é de 48-bits, portanto o máximo endereçável é 2 elevado a 48 que seria 256 Terabytes. 2-bits fazem muita diferença, porque é o quadrado do quadrado, então na Intel você conseguiria endereçar no máximo 4 Terabytes.

Isso é ordens de grandeza maior do que em computadores 32-bits da geração passada. E não, 64 não é o dobro de 32-bits, é o quadrado. Lembrando que 32-bits significa 2 elevado a 32, que limita o endereçamento a no máximo 4GB de RAM. Se você tinha computador nos anos 2000 vai lembrar que não dava pra colocar mais que isso de RAM. Se você chegou a ter acesso a servidores com Windows Server 2003 Service Pack 2 Datacenter Edition dava pra ter até 64 GB.

Isso porque havia um truque da Intel em processadores 32-bits chamado PAE ou Physical Address Extension, e dependendo da configuração de hardware e sistema operacional, dava pra ir acho que até um máximo de 128 Gigas mesmo num computador 32-bits. Na prática, os processos num sistema desses continuavam só enxergando 4GB mas era possível ter mais processos ativos porque com PAE você podia swapar alguns processos pra essas áreas extendidas de memória e evitar usar swap de disco por exemplo.

Tudo isso dito, vamos tentar explicar mais sobre como o sistema operacional enxerga essa memória. Até agora sabemos que a memória é como um livro, com linhas de endereço no índice, cada endereço representando uma página do livro, com o endereço máximo dependendo se o computador tem barramento de 48-bits, 42-bits, 36-bits, ou 32-bits e assim por diante. O sistema operacional pode ler e escrever em qualquer endereço de memória. Num sistema operacional muito simples, tipo um Basic num Commodore 64, dava pra acessar diretamente esses endereços pra escrever ou ler.

Na verdade existe um porém, você poderia achar que seu programa tem acesso a tudo do endereço 0000 até o endereço FFFF por exemplo, num computador de 16-bits. Mas quando você boota um Commodore, apesar dele poder endereçar 64KB, porque tinha barramento de 16-bits, você vai ver que ele diz que só tem 38.911 bytes livres. Isso porque como em qualquer máquina, nem todos os endereços estão livres pro usuário.

Um pedaço dos endereços é reservado pra kernel, outro pedaço são funcionalidades como endereços de I/O, outro pedaço no caso do Commodore é o próprio interpretador Basic, e o que sobra é onde você pode carregar seus programas. O programa Basic só pode carregar a partir do endereço $0801 até o $A000, se você fizer a conta, A000 é o decimal 40,960 e 0801 é o decimal 2049. Então 40,960 menos 2049 são os 38911 bytes que o sistema declara como disponível.

Isso era na época em que tudo usava endereços reais, até o fim dos anos 80, antes dos processadores Intel 80286. Os computadores modernos a partir dos anos 90 tem o tal modo protegido e memória virtual. Quando o sistema operacional carrega seu programa ele dá pra ele um índice virtual, que vai do endereço virtual 0000 0000 até o FFFF FFFF. Só que o endereço virtual 0000 0000 desse processo, na memória real pode ser um endereço nada a ver, faz de conta, DCBA 9876, mas seu processo não tem idéia disso, só o sistema operacional sabe mapear entre os dois.

Daí você carrega outro programa, ele também ganha outro índice virtual que vai de 0000 0000 até FFFF FFFF só que o 0000 0000 desse programa mapeia pro endereço real BA98 7654 por exemplo. Ambos os programas enxergam o 0000 0000 mas cada um aponta pra um lugar diferente. E por isso os dois programas são ditos isolados, porque não importa que endereço eles tentem apontar, nunca vai ser em cima do endereço do outro processo porque o sistema operacional garante que um endereço real usado não é apontado por mais do que um índice virtual.

A grande vantagem disso é que você pode ter 10 programas, cada um “enxergando” 4 GB de memória inteiro só pra ele, ou seja endereços virtuais de 0000 0000 a FFFF FFFF mas seu computador real pode ter bem menos memória que isso. Na prática a maioria dos programas não vai usar toda a memória disponível, então o sistema operacional compartilha a memória real entre os diversos programas até acabar a memória real. Então seu processo tem só a ilusão de que tem toda essa memória. E o que acontece quando acaba 100% da memória real do sistema, ou seja, toda a RAM e todo o swap no HD? Antigamente seu computador ou ia ficar lento ao ponto de ser inusável, ou programas iam começar a crashear por falta de memória, ou alguma combinação disso.

Num sistema operacional de smartphones você já sabe o que acontece. Num iPhone ou qualquer Android moderno, você vai abrindo programas e não precisa fechar. Quando acaba a memória, os programas menos usados ou os abertos à mais tempo e sem uso, fecham sozinhos, dando espaço pra abrir mais programas. Num Android, que é um derivado de Linux, e mesmo iOS que é derivado de BSD UNIX, existe o OOM Killer ou Out of Memory Killer. Todo Linux tem isso, e é isso que evita que seu celular fique sem memória. É por isso que falamos que em celulares modernos você não precisa ficar fechando manualmente os programas, o OOM Killer vai fazer isso por você.

Voltando aos seus processos, o sistema operacional começa dando zero bytes de memória real pro seu processo. À medida que o processo vai pedindo memória, ele vai alocando memória real, sob-demanda, e mapeando os endereços reais pra endereços virtuais que o processo enxerga. E quando o processo vai devolvendo memória ou quando você fecha o programa, esses endereços reais voltam a ficar disponíveis pra outros processos poderem usar. Se você simplesmente abre programas de Linux como ps, top ou htop ou o Task Manager no Windows, vai quanto cada programa está alocando. Pra simplificar digamos que você tem 2 programas, cada programa você vê ocupando 1.5 GB. Ou seja, cabe tudo em 3 GB de RAM. Daí na sua cabeça você acha que porque tem 4 GB de RAM, tem memória de sobra, certo? Errado. Um desses programas pode estar muito perto de crashear.

Pra simplificar, vamos considerar ainda um computador de 32-bits nesse exemplo. Eu disse que seu programa enxerga do endereço virtual 0000 0000 até FFFF FFFF. Mas ele Não vai poder usar tudo isso. Num Windows 32-bits de mais de 10 anos atrás, você começa tendo só 2 GB disponível, do começo até a metade, ou seja de 0000 0000 até 7FFF FFFF. Daí pra cima é reservado ao sistema operacional. Lembra como eu falei no Commodore 64 que partes dos endereços são reservados pro Kernel, Basic, I/O e outras coisas? Mesmo coisa em sistemas operacionais modernos. Mesmo num Linux, ele vai reservar de 512MB a 1 GB de endereços. Num Windows Server você podia bootar com um flag especial e compilar os programas com a flag /largeaddressspace pra permitir usar um pouco mais além da metade dos endereços. Não quer dizer que o Windows está realmente usando 2 GB, mas sim que os endereços estão reservados pra ele usar.

Ou seja, num computador 32-bits, apesar de poder mapear até 4GB e mesmo se você tivesse 4GB reais de RAM, nenhum processo seu iria conseguir usar mais que 2GB. Hoje em dia você tem problemas de memória em computadores 64-bits? Imagina quando a gente tinha que programar servidores em 32-bits. Pior ainda, imagina como era na época dos 16-bits. Saber usar memória é até hoje o que diferencia um amador de um profissional de verdade. Pra dar um exemplo, lembram do episódio que eu falei de SAP? Nessa época, acho que era 2003, eles tinham um servidor de aplicações Java, o In-Q-My, ainda em beta, pra carregar aplicações J2EE pesadas como o Enterprise Portals. No boot do Java com os programas SAP por cima, ele carregava tranquilamente um giga e meio.

Se você fosse amador, pensaria, “ah, sussa, máquinas 32-bits carregam até 4GB então tem sobrando”. Se você fosse intermediário pensaria “puta que pariu, só mais 500 Mb e vai bater no limite de 2 GB”. Porém, eu não conseguia fazer esse servidor subir, ele crasheava por falta de memória. Mas eu descobri porque. Lembram que eu falei que o sistema operacional usa parte da memória do processo? Seus programas reusam muitas bibliotecas que o sistema operacional compartilha com os processos, então você tem endereços virtuais que mapeiam pros endereços reais dessas bibliotecas do sistema, incluindo muitas DLLs que todos os processos usam como a LibC da Microsoft ou o Foundation Classes. Pra você entender, o sistema não está duplicando essas bibliotecas em todos os processos, mas os endereços virtuais dos processoa mapeiam pras mesmas bibliotecas. Os processos são isolados, então pra eles enxergarem alguma coisa de fora, os endereços virtuais precisam mapear pra esses recursos compartilhados.

Ainda pensando de forma ingênua, você imaginaria que essa bibliotecas carregam bonitinhas sequencialmente uma atrás da outra, sem desperdiçar nenhum endereço. Lógico que não, não necessariamente. E de fato, algumas dessas bibliotecas apareciam com endereços na primeira metade dos endereços virtuais do processo, ou seja, em cima dos 2 GB reservado pro seu programa. Isso se chama fragmentação de memória. A JVM precisava alocar memória contígua no boot, ou seja com endereços sequenciais, sem buracos no meio. Aliás, alocação de memória sempre aloca pedaços inteiros, sequenciais. Como a memória estava fragmentada, não havia um trecho sequencial limpo de 1.5 GB. E agora?

Felizmente, no Windows você tem como fazer rebase das DLLs, pense mais ou menos, como defragmentar a memória. Exatamente isso, você pode mudar o endereço de carregamento dessas DLLs. Como são bibliotecas dinâmicas, o binário não mapeia um endereço fixo, então você pode mudar elas de lugar. E foi assim que eu consegui subir os programas Java que eu precisava. Quem já trabalhou com servidores 32-bits e programas realmente pesados de Java, com certeza esbarrou com isso, é um problema sabido desde o ano 2000 pelo menos. E pense que em 2003 o site StackOverflow ainda não existia.

Aceite este fato: a memória total do seu sistema não está à disposição do seu processo. Felizmente em máquinas 64-bits de hoje, você tem muito mais que 2 GB de endereços virtuais disponíveis, então pelo menos esse problema já não afeta mais. Tanto que mesmo na época uma das soluções que a SAP falava era: migre pra máquinas 64-bits. Espero que em 2019 ninguém mais esteja usando máquinas 32-bits. O segundo fato que você precisa aceitar: memória fragmenta. A razão é simples. Digamos que sua memória total seja de 10 kb, dividida em chunks ou pedaços de 1 kb. Você começa alocando 3 variáveis de 1 kb cada. Digamos que o alocador aloque os primeiros 3 chunks. Agora digamos que você resolva liberar a segunda variável. Você ficou com 1 kb ocupado, 1 kb livre, e o terceiro kb ocupado.

Agora seu programa resolve que precisa de uma variável de 2 kb. Como o espaço vago do meio não cabe os 2 kb, e repetindo, grave essa regra: o alocador sempre precisa alocar pedaços contíguos, com endereços sequenciais. Então ele só vai ter espaço depois do terceiro chunk. Agora você ficou com um espaço fragmentado. Pense numa escala maior, com milhares de alocacões e desalocações, sua memória vai ficando com buracos no meio do caminho e toda vez que você precisa alocar mais memória do que cabe nesses buracos ele precisa ir lá pra frente. Então um programa que internamente ele enxerga que está alocando só 4 KB na verdade está potencialmente bloqueando 5 KB nesse momento. Fragmentação é uma das coisas que alocador de memória quer evitar.

Quem faz esse trabalho é o alocador de memória. No caso do Linux, mais especificamente da biblioteca glibc, você tem o famoso “malloc”. Eu expliquei nos episódios anteriores que no Linux você pode escolher entre diferentes schedulers de threads como o CFS que hoje é o padrão ou outros como o BFS dependendo de pra que vai usar a máquina. Alocador de memória também, você pode escolher.

O malloc do glibc na verdade é o ptmalloc2 que por sua vez é baseado no dlmalloc, DL porque quem criou essa versão foi o professor Doug Lea. Um alocador de memória é um conjunto de algoritmos e estruturas para manter metadados dessa memória. Um alocador moderno tem vários de problemas importantes pra resolver. Primeiro de tudo, alocar memória precisa ser rápido, estamos falando de frações de milissegundos, ou seja, nanossegundos, porque é uma operação que acontece o tempo todo. Um ptmalloc2 leva uns 300 nanossegundos por operação. Mas não adianta ser rápido se ele “vaza” memória ou seja, usa mais memória do que o necessário, em parte porque não foi inteligente e deixou a memória fragmentar demais. Existem várias syscalls mas em C você normalmente chama pelas funções malloc, passando um tamanho que precisa alocar, e a função free que sinaliza ao sistema operacional que esse endereço pode ser liberado quando puder.

Quando você chama a função malloc, o único parâmetro que passa é quanto de memória quer alocar, e ele devolve como retorno o endereço que ele achou onde cabe um pedaço de dados do tamanho que você pediu. Se você tiver pouca RAM e vários processos fragmentando memória, você vai notar que teoricamente era pra caber mais programas na RAM mas por alguma razão falta memória. Um programa mal escrito pode estar alocando mais memória do que deveria, e não devolvendo. E mesmo devolvendo o sistema ainda pode não ter colocado essa memória à disposição.

Pra piorar, eu já expliquei sobre concorrência e paralelismo e todos os problemas que você enfrenta ao lidar com threads reais, lembra? Existe um outro problema: e se várias threads pedem ao alocador pra reservar memória ao mesmo tempo? Acabamos de criar uma contenção, ou seja, um gargalo. Se o alocador não suportar threads corretamente, o que acontece é um lock do alocador e cada thread tendo que esperar o anterior receber seu pedaço de memória. Na verdade o tal ptmalloc que eu falei é um fork do antigo dlmalloc justamente pra suportar threads.

Só de entender o problema de fragmentação e agora o problema de threads, você deveria começar a entender que fazer uma chamada em malloc não é um troço simples. Pra começar a forma de suportar threads é dividir o índice de endereços disponíveis no que o malloc chama de Arenas. Num Linux 32-bits o padrão é 2 vezes o número de cores. Então num computador quad-core com hyper threading você pode ter até 8 threads e por isso até 16 arenas. De forma simplista se todas as threads pudessem ter qualquer endereço do espaço de 32-bits, o alocador ia precisar criar um lock global e as threads perderiam totalmente sua performance e ficaram mais tempo em fila esperando o alocador do que realmente trabalhando em paralelo. Mas se você tem 4 cores e disser que do endereço 0000 0000 a 3FFF FFFF é da thread 1, do endereço 4000 0000 até 7FFF FFFF é da thread 2 e assim por diante, agora elas não compartilham endereços e você não precisa de um lock. Lembra que eu dei a dica já que a forma de você não ter que lidar com mutexes e locks em concorrência é não compartilhar nada entre as threads?

Agora as Arenas ainda são sub-divididas em Heaps ou montes. Seu programa quando executa pode armazenar dados na stack ou pilha ou no heap. A pilha é literalmente a pilha de execução, seu programa carrega as instruções na stack e pequenas quantidades de dados cabem no stack, se usar demais ou de forma errada vai ter o famoso Stack overflow. Se a memória fosse uma página de papel, pense na stack crescendo de cima pra baixo e o Heap começando de baixo pra cima. Não é exato mas é uma forma de visualizar. Na prática seu programa vai armazenar a maior parte dos dados que precisa pra trabalhar no Heap e na stack podem ficar coisas como valores constantes ou coisas assim.

Uma heap é ainda sub-dividido em múltiplos chunks. Podemos ter chunks muito pequenos de até 80 bytes, chunks de até 512 bytes e chunks grandes de mais de 512 bytes, que é como o ptmalloc organiza. Podemos agrupar os chunks menores em caixas ou bins, no caso os de até 80 bytes no que se chama Fast Bin, os de até 512 bytes num Small bin e os maiores que 512 bytes num Large bin. Por que isso? Pra minimizar fragmentação. Você fragmenta a memória mais rápido se tentar misturar pedaços grandes no meio de pedaços pequenos. É melhor agrupar pedaços de tamanhos similares mais próximos, porque se você desaloca um chunk de 16 bytes no Fast Bin, quando precisar alocar 16 bytes de novo, é mais fácil procurar direto no Fast bin. E se tentar alocar alguma coisa de 1 MB é melhor ir no Large Bin. E no começo você aloca poucos bins, e vai alocando mais à medida que precisar.

Lembre que toda essa organização ainda não está na prática “usando” a memória ainda. Está só criando estruturas que representam caixas de endereços. Daí a alocador vai pegar desses bins, dependendo de que thread está pedindo e que tamanho de memória pediu. Se você tem boa intuição está entendendo que seu processo sempre vai usar um pouco mais de memória do que ele realmente precisa por causa dessa organização e estruturas de suporte que o alocador precisa. Quanto mais tempo um processo fica vivo trabalhando, mais essas atividades de alocar e desalocar memória podem is expandindo, crescendo o heap e causando leaks se o programa for mal escrito, causando fragmentação demais.

Mas o pior não é isso, eu disse que Arenas são divisões para evitar contenção de lock de threads. E se sua aplicação inicialmente carregar 300 MB de estruturas na primeira arena. Mas agora numa segunda fase outra thread precisa trabalhar essas estruturas mas ela vai usar outra arena? Os 300 Mb vão ser copiados de uma arena pra outra porque o ptmalloc não suporta mover memória entre arenas. De repente seu programa inicia usando o dobro do que ele realmente precisa. Esse é um problema real relatado por engenheiros do Google e por causa disso eles criaram um outro alocador de memória, chamado de tcmalloc, que justamente permite cachear dados entre threads e por isso se chama Thread-Caching malloc ou tcmalloc.

Se algum dia esbarrar num tutorial que fala de tcmalloc, lembre que ele é do Google. Ele é mais rápido que o ptmalloc2 do glibc, chegando a ser até 6 vezes mais rápido e usando estruturas de dados mais eficientes, então também usando bem menos memória no geral. E como o tcmalloc implementa as mesmas funções POSIX como o malloc(), free(), calloc(), realloc() e outros. O Google mesmo diz que ele não é necessariamente 100% compatível e alguns programas podem ter problemas, mas na grande maioria, você consegue carregar o tcmalloc e seus programas devem funcionar mais rápido, usando menos memória e, se for um programa que fica de pé por dias seguidos como um servidor de aplicação, ele também deve tender a alocar menos memória no geral.

Tcmalloc foi criado mais de 10 anos atrás, e na mesma época, com o Firefox 3 o alocador de memória do Windows da época fragmentava demais, ou seja, o navegador tendia a crescer no uso de memória como eu já expliquei que acontece, numa ordem de pelo menos 20% ou mais de desperdício. Se o processo usar 1 GB ele vai desperdiçar 200 Mb só em fragmentação. Nessa época um engenheiro chamado Jason Evans queria resolver o problema de gerenciamento de memória de uma linguagem experimental que ele estava criando e pra isso fez um novo alocador. Esse alocador passou a ser usado no Firefox e foi um dos motivos de porque Firefox no Windows conseguia desperdiçar menos memória. Com o tempo esse alocador foi adotado pelo Facebook porque seus servidores também estavam desperdiçando muita memória. Esse alocador foi devidamente nomeado de jemalloc por causa de Jason Evans. Com o tempo o jemalloc se tornou mais rápido, mais robusto e confiável e desperdiçando ainda menos memória que o tcmalloc do Google. Inclusive acho que ele é mais compatível com o ptmalloc2 e você poderia trocar o alocador do seu Linux por jemalloc e tudo vai funcionar melhor.

De curiosidade, a linguagem Rust, assim como o Firefox, são da mesma fundação Mozilla e por isso o Rust desde o começo sempre usou o jemalloc, que é um dos motivos de porque ele é eficiente com memória. A grande maioria das outras linguagens ainda usam o ptmalloc2 do glibc. Linguagens como Python, Ruby também se beneficiam se você fizer o interpretador usar o jemalloc. Vou deixar nas descrições abaixo como fazer isso no Ruby.

Agora, isso vale pra linguagens que ativamente usam o alocador do sistema operacional. C, ou C++, ou Rust que já falei, todos que compilam pra binários nativos vão usar o ptmalloc2 ou o jemalloc hoje em dia. Pra variar, o Go não usa nenhum que já existia. Um ptmalloc2 usa funções de mais baixo nível ainda do sistema operacional como mmap, madvise, munmap, sbrk. O Go reinventa a roda e implementa seu próprio alocador meio inspirado, claro, no tcmalloc do Google. O Go também divide a memória em várias arenas. Esse conjunto de arenas é o que ele chama de Heap. Ele organiza as arenas em páginas de 8kb. Ele também divide esses blocos em bins pra tamanhos diferentes de chunks mas no caso do Go os nomes são diferentes ele tem um Tiny pra objetos menores que 16 bytes, um Small pra objetos menores que 32 kb e um Large pra coisa maiores que 32 kbytes. Esse valores dependem da versão do Go, então não decorem isso, só lembrem que o alocador agrupa blocos de tamanhos similares, como o ptmalloc2 já fazia também. E que é uma das estratégias pra diminuir a fragmentação.

Escovamos bits demais por hoje, vou ter que dividir esse episódio em duas partes de novo. Neste episódio tivemos mais noção de como o sistema operacional realmente organiza a memória, as diferenças de arquiteturas 32-bits ou 64-bits, e os diferentes tipos de alocadores. E como um bônus de termos já discutido sobre threads nos últimos episódios, vimos agora como eles afetam o gerenciamento de memória também. Na semana que vem vamos falar sobre os famosos garbage collectors, que muita gente ainda acha que é mágica, mas na prática não tem idéia de como funcionam.

parte 2

Recapitulando, no episódio anterior falamos um pouco das diferenças de alocadores como o ptmalloc2, tcmalloc do Google e o jemalloc que o Facebook usa. No fim, todo alocador vai querer resolver o problema de fragmentação, o problema de contenção de threads, o problema de cache entre threads. E agora temos os garbage collectors. Pense em garbage collectors como sendo um alocador em user-land. Ele faz syscalls pro sistema operacional, como o malloc() pra pedir blocos grandes de memória e internamente gerencia como esse pedação vai ser usado. Python, Ruby, Erlang, Javascript, .NET, Java e por consequência toda linguagem que roda em cima de Java como Scala, Clojure, Kotlin, usam o mesmo garbage collector da JVM. Assim como schedulers the coroutines em user-land são mais rápidos como eu já expliquei em episódios anteriores, um alocador em user-land também é mais rápido do que fazer syscalls pra funções no kernel-space como mmap ou munmap que é o que funções como malloc() fazem.

Todo garbage collector faz algumas coisas meio básicas. Não dá pra explicar no detalhe como o collector de cada linguagem funciona, mas vamos ver os conceitos comuns. Pra começar, existem linguagens que lidam com ponteiros, ou seja os tais endereços virtuais diretamente, e os que lidam com referências que são como se fossem atalhos pros endereços. Garbage collectors existem principalmente porque nós programadores somos muito ruins em liberar memória manualmente.

Se você teve que lidar com C, sabe que precisa chamar a função free() pra toda memória que deu malloc() antes. Esqueça de dar free e seu programa vai começar a ter leaks ou vazamentos e vai ficar crescendo sem parar. Parece super simples, mas toda regrinha desse tipo você pode ter certeza que vai esquecer. Basta esquecer de digitar uma linha ou apagar uma linha do código com free por acidente. Ou pior, se você der free numa memória que ainda não era pra dar free, vai ficar com um Invalid Pointer por exemplo e seu programa pode crashear. Quando você adiciona classes e objetos em cima do C e gera um C++ a coisa fica pior ainda porque você vai passando ponteiros de objetos pra outros objetos numa árvore gigante de dependências. Quando que você pode destruir um objeto sem deixar outro que dependia dele corrompido?

Uma das formas de resolver esse problema é adicionar a todo objeto um contador. E toda vez que você passar esse objeto pra outro objeto ou função, incrementar o contador. Quanto mais objetos apontarem pra esse primeiro, maior vai ficando o contador. À medida que os objetos que apontam pro primeiro também forem sendo liberados, o contador vai decrementando. E quando o contador voltar a ser zero, daí você sabe que ninguém está mais usando e pode liberar esse objeto. Como estamos falando de contagem de referências, você vai ouvir falar nessa estratégia como Reference Counting. Linguagens como Python e Objective-C e, por consequência, Swift usam essa estratégia.

No caso do Objective-C ele vai um passo além, ele inclui uma classe chamada NSAutoreleasePool, um tanque de alocação. Você instancia um pool e à medida que vai alocando novos objetos esse pool fica sabendo deles. Diferente de um garbage collector, em Objective-C você manualmente instância quantos pools quiser, e tem que manualmente chamar release no objeto que, diferente da tal função free(), vai só decrementando o contador de referência. E quando quiser pode chamar o método release do pool, que por sua vez dá free de verdade nos objetos com contador zerado.

Nas versões mais novas do Objective-C e no Swift, coisa da última década, a Apple adicionou o ARC ou Automatic Reference Counting. Na prática ele vai cuidar de adicionar o código pra dar release do contador usando algumas dicas no seu código, durante a compilação. Vale a pena estudar em detalhes como o ARC funciona se você faz programas pra iPhone ou Mac, mas 90% dos casos você não se preocupa tanto com a memória e vai precisar saber mais quando tiver bugs ou quiser otimizar o uso de memória, por exemplo forçando a limpeza do pool mais vezes. Digamos que você está lendo um arquivo com 500 linhas, em vez de fazer o loop e só no final limpar tudo de uma vez, você pode ir dando release no pool manualmente a cada 50 linhas por exemplo e evitar que o sistema aloque mais memória do que precisa. Se você desaloca memória com muita frequência, vai usar mais CPU e diminuir a performance geral, por outro lado vai consumir menos memória. Se você desalocar com pouca frequência, aumenta a performance, mas também aumenta o uso de memória.

Um problema de contagem de referências é o caso de reference cycle ou seja, você tem objeto A com uma referência pro objeto B daí no B fazemos uma referência pro objeto A, agora ambos os contadores de A e de B nunca vão ser menores que 1, e o pool não vai conseguir desalocar a memória desses objetos mesmo que ninguém mais esteja usando, porque os contadores não são zero. Por isso em Objective-C e Swift você tem a diferença entre strong references que são as referências normais. Mas também temos referências weak e unowned, nenhuma das duas incrementa o contador e isso quebra o ciclo de referências. No caso a referência do objeto B pro objeto A poderia ser declarada como Weak e daí o contador de A chegaria a zero normalmente e ambos poderiam ser limpos. Novamente, você precisa ler em mais detalhes a diferentes de uma referência Weak e Unowned que vou deixar nas descrições abaixo. Linguagens com contadores de referência como Python costumam ter a opção de referências fracas ou Weak.

Controlar os contadores de uso de cada objeto é uma forma de você não precisar manualmente liberar objeto a objeto quando não precisa mais. Basta ter um controlador como o NSAutoreleasePool que consegue limpar todo objeto com contador zerado de uma vez só. Em C++ temos smart pointers que funcionam mais ou menos parecido. Um garbage collector muito simples poderia simplesmente ser uma thread que, de tempos em tempos, manda limpar esse pool. Seria uma forma rudimentar de garbage collector. Mesmo Python trabalha com contador de referências também, mas seu collector é mais sofisticado do que só isso.

Eu falei que em Objective-C você pode criar quantos pools quiser e controla eles manualmente. Outra forma é esse Pool ser tipo uma estrutura global e única, é como funciona em Ruby e em Java e acho que em .NET também. Pense numa lista gigante, onde sempre que precisarmos de memória pedimos pro alocador que vai adicionando nessa lista também, catalogando todos os elementos que vão sendo alocados. Toda vez que você passa a referência desse elemento pra outro objeto, ele vai criando uma árvore de dependência. Objeto A pode ter referências pros objetos B e C, e o objeto C tem referência pro objeto D. Depois de algum tempo, a variável que aponta pra A deixa de existir, por exemplo o método retornou ou você passou nulo pra variável e então ninguém mais aponta pro objeto A.

Com o tempo vários objetos vão ficando nesse estado, onde ninguém mais aponta pra eles a não ser a lista global que falei. Então de tempos em tempos um processo especial chamado collector, numa thread no interpretador ou na VM, poderia ir vasculhando essa lista por todos os objetos que ninguém mais está apontando e ir só marcando. Depois um segundo processo poderia ir na mesma lista, elemento a elemento, desalocando os objetos que foram marcados. Então se a lista tinha 1000 objetos, cada um ocupando 1 slot da memória, na fase de marcação 500 objetos foram marcados como não usados, e na fase de limpeza esses são liberados os 500 slots pra uso. Sendo ultra simplista é isso que chamamos de um garbage collector Mark and Sweep. É um collector de duas fases, onde em cada fase ele precisa vascular elemento a elemento da lista de alocação, fazendo um mark na primeira fase e sweep na segunda.

Porém, quando eu falo desalocar não quer dizer devolver memória pro sistema operacional. Lembra do problema de fragmentação? Se eu ficar devolvendo memória toda hora, muito rapidamente a memória do processo vai ficar fragmentado. Linguagens como Ruby ou Java começam pedindo um certa quantidade de blocos de memória pro sistema operacional. No caso de Ruby tem o equivalente de chunks que ele chama de slots, cada slot tem um tamanho fixo de 40 bytes onde cabe toda a estrutura que define um objeto de Ruby.

Então ele só lida com slots fixos de 40 bytes. Um conjunto desses slots é chamado de Heap. Não confundir com o Heap do malloc de C, é o mesmo nome mas tanto Java quanto Ruby chamam conjuntos de slots ou chunks de Heaps. No caso do Ruby existem várias variáveis de sistema pra configurar o alocador, como por exemplo o RUBY\_GC\_HEAP\_INIT\_SLOTS, que por padrão o valor é 1000. Um heap de Ruby costuma ter 16 kbytes. Ou seja, ele vai alocar de 40 bytes x 1000 que seria quase 40 kbytes ou pelo menos 2 heaps.

Se acabar os slots disponíveis, o alocador do Ruby vai pedindo pra alocar mais heaps, vários slots de uma só vez. Digamos que ao carregar seu programa ele precisou de 10 mil heaps, ou seja, uns 156 megabytes. Agora digamos que depois de rodar por um tempo, uns 500 mil slots de objetos foram marcados como não usados mais. Na fase de Sweep você imaginaria que ele vai devolver 500 mil x 40 bytes que dá quase 20 megabytes, de volta pro sistema e em vez de ocupar 156 megabytes ele vai passar a usar 136 megabytes. Mas não, no final do Sweep seu programa vai continuar usando os mesmos 156 megabytes. Por que? Porque ele só vai liberar os slots da lista global, mas a memória de verdade continua alocada. Quando você criar novos objetos nos slots livres, ele vai reusar e escrever em cima do mesmo trecho de memória de antes.

De cara temos estratégias diferentes entre o Heap de uma linguagem como Ruby ou o Heap de uma linguagem como Java. No caso de Java, todos os objetos ocupam espaço no Heap. No caso de Ruby só a estrutura e dados muito pequenos ocupam os slots de 40 bytes. Mas um array ou um string costuma ser muito maior que 40 bytes, então o Ruby chama a syscall malloc do sistema operacional e guarda o endereço alocado no slots de 40 bytes apontando pro espaço fora do Heap. A vantagem do Ruby é que todo objeto tem o mesmo tamanho dentro do Heap e isso ajuda a não fragmentar tanto, mas ele depende do alocador do sistema operacional pros blobs de dados maiores. No caso do Java ele não tem que se preocupar com o sistema operacional, porque ele usa só sua memória interna do Heap, mas outro lado isso vai causar fragmentação interna e ele tem uma forma de evitar isso, que vamos ver já já. Mas só pra entender que diferentes collectors organizam seus dados internamente de formas diferentes.

Do ponto de vista de programação, se você ainda não entendeu, alocar memória desnecessariamente vai aumentar o tamanho dos heaps ou a quantidade de heaps, dependendo do alocador, e isso vai bloquear memória do sistema operacional e causar fragmentação desnecessária também. Você não pode achar que vai alocar 1 giga de dados, devolver esses 1 giga, e vai continuar tendo bonitinho 1 giga de novo sobrando. Você vai fazer o alocador ter que trabalhar muito pra evitar a fragmentação e se seu sistema tiver pouca memória geral, provavelmente não vai conseguir ter esse espaço livre de volta.

Exemplos de má programação é o famoso caso de abrir um arquivo gigante de vários gigas na memória, ou conectar num banco de dados e não fazer o select com os critérios de where certos e acabar puxando muito mais linhas da tabela do que precisava. Tudo isso vai ocupar espaço, o alocador vai ter que esticar os heaps, mais memória do sistema vai ser consumido, e não necessariamente ele vai liberar essa memória de volta pro sistema. Mesmo depois que o collector limpar os slots ocupados por esses dados, você ficou com um heap inchado e fragmentado. E se seu servidor for pequeno, você vai começar a ter problemas de memória mesmo que teoricamente tenha “memória sobrando” ainda.

Se seu programa rodando em produção tem leaks de memória, normalmente são dados que vem da rede, como dados do banco de dados, ou processamento de arquivos grandes. O ideal é sempre seu código tentar usar memória até um limite. E quando precisa puxar dados externos, ter algumas checagens pra ver se não vai usar memória desnecessariamente. Tipo checar o tamanho de um arquivo antes de carregar ele inteiro na memória. Se for arquivos pequenos de alguns kbytes, dá pra ler tudo e armazenar numa variável de string. Mas se for um arquivo de gigabytes, é melhor abrir como um Stream e ir lendo pedaços do arquivo de cada vez. Ou quando precisa puxar dados de uma tabela num banco de dados, usar coisas como um limit no select pra limitar quantas linhas vai puxar de uma só vez. Famoso caso que o amador fala que na máquina dele funciona, só que na máquina dele ele só testou com arquivos pequenos e com tabelas com 10 linhas de teste. Por isso quando vai pra produção tem esse comportamento. Isso porque é um programador que não tem noção de gerenciamento de memória.

Voltando, outro problema é se seu programa é grande e aloca milhões de objetos. Toda vez que o processo de Mark e Sweep rodar, eles precisam passar por todos eles, um por um, toda vez. Mesmo que só leve uma fração de milissegundo pra marcar e sweepar cada objeto, ainda assim vai levar dezenas de segundos pro processo todo se contar milhões de objetos alocados. É muita coisa. Pior ainda, toda vez que você vai marcar os objetos ou limpar, precisa pausar o programa todo, é o que chamamos de Stop the World. Isso porque você tem que garantir que o programa rodando ao mesmo tempo que o collector não se corrompa tentando acessar um objeto que acabou de ser desalocado. Só que se você pausar seu programa por muitos segundos, ele vai ficar muito lento.

Ou seja, carregar dados desnecessários não só desperdiça memória como desperdiça performance. Agora você só tem 2 escolhas. A primeira é rodar poucas vezes o Mark e Sweep, mas aí você arrisca consumir muita memória e fazer seu heap crescer demais já que sempre que precisa de mais memória e não tem slots pra reusar ele vai pedir mais pro sistema operacional. E eu já disse que toda memória que ele aloca do sistema ele não devolve mais ou vai demorar pra devolver pra evitar fragmentação, daí seu processo vai crescer mais do que devia. A segunda escolha, se você quer evitar isso então tem que rodar mais vezes o Mark e Sweep só que daí a performance geral do programa vai ficar lento. Ou seja, agora você tem que escolher entre performance ou uso de memória.

Pra resolver esse problema é que existe a idéia de Generational Garbage Collectors. Ele vai dividir a memória interna do interpretador ou da máquina virtual em 2 ou mais gerações. Java divide seus heaps em grupos chamados Young, Old e Permanent (por isso você vira e mexe vê configuração em tutorial de JVM pra configurar coisas como o Permsize que é o tamanho desse espaço permanente). O garbage collector G1GC novo do Java chama de Eden, Survivor e Old. Python chama só de geração zero, um e dois.

A idéia vem de uma hipótese chamada Weak Generational Hypothesis. Em linhas gerais essa hipótese diz que objetos novos tendem a morrer jovens e objetos velhos tendem a ficar ativos por muito tempo. Parece óbvio quando falamos assim, né? Eu já expliquei como o sistema operacional divide a memória em arenas por exemplo, pra evitar contenção de threads. Agora vamos dividir mais ainda pra evitar passagens desnecessárias por objetos que não vão morrer tão cedo.

O problema que um generational garbage collector quer resolver é diminuir ao máximo as pausas da fase de mark and sweep. A forma de fazer isso é evitar ficar olhando objetos que nunca vão ser limpos. Por exemplo, objetos singleton que seu programa carrega no começo e vai usar até morrer. Num Ruby, toda classe é um objeto. Mas as classes raramente precisam ser limpas, então pra que perder tempo olhando pra elas toda vez? Em resumo, na primeira vez que o mark passar, ele vai ter que olhar tudo. Objetos que sobrevivem a múltiplas passagens da fase de mark podem ser promovidos da geração Nova pra geração Velha e se sobreviverem mais algumas passadas de mark, podem ser promovidas pra geração permanente.

Podemos dividir as fases de Mark em Minor e Major. O Minor só olha pros objetos mais novos e ativos na geração Nova. O Major olha também pra geração Velha pra ver se algum precisa ser limpo. Mas uma ver promovido pra permanente, podemos esquecer desses objetos. Digamos que um programa normal tenha 100 mil objetos, e que depois de algumas passadas da fase de Mark 50 mil deles tenham sido promovidos pra geração permanente. Os outros 50 mil são objetos que morrem, e novos que são criados e na média ficam sempre uns 20 mil na geração nova, uns 30 mil na geração velha. A performance total da fase de mark é pelo menos o dobro, porque você no mínimo vai passar por metade dos objetos toda vez. E tende a ser mais rápido que isso porque você passa menos vezes na geração velha também. Então é um ganho enorme de performance.

Numa linguagem como Ruby ou Python, esses segmentos como eu disse são só listas com ponteiros pros slots ocupados. Como por baixo é tudo estrutura em C temos endereços e como as extensões em C podem precisar do endereço exato de determinada estrutura não podemos mover os dados de lugar.

Eu menciono isso porque em Java temos uma coisa extra. Java não lida diretamente com ponteiros, ele trabalha com referências no nível da linguagem. Uma variável em Java aponta pra uma referência e a referência, por baixo dos panos, é indexada num endereço. Mas o endereço pode mudar. Diferente de Ruby ou Python, em baixo nível o alocador pode mover o objeto de um slot pra outro nos heaps. Mover fisicamente vai mudar o endereço e aí eu mudo na referência. Mas no alto nível a variável no Java só vê a referência então pra ele nada mudou. Uma coisa é eu dar um produto na mão de uma pessoa. Agora não posso mais tirar o produto da mão dela sem ela reclamar. Outra coisa é eu dar um vale produto pra pessoa. Agora no estoque tanto faz se eu mudar o produto de lugar, o que vale é a referência.

Isso é importante porque na JVM quando acontece a fase de promover os objetos da geração jovem pra geração velha, em vez de só mover o ponteiro de um segmento pra outro de uma lista, na verdade eu estou movendo o objeto de um slot da geração jovem pra um slot da geração velha, fisicamente falando. Estou copiando de um slot pro outro e depois liberando o primeiro slot. Então fisicamente na memória, os heaps de Java tendem a ser menos fragmentados que num Python ou Ruby.

E os objetos de Java podem alocar tudo dentro do objeto em vez de usar a estratégia de dividir o objeto em 2 partes, sendo uma estrutura de tamanho fixo e o resto dos dados alocados fora dos heaps via malloc do sistema operacional. Nesse caso os objetos de Java tem tamanho variável, o que causaria fragmentação como eu já falei mas como ele pode mover os objetos de lugar e ir defragmentando a cada passada do collector, então evitamos fragmentação. É como se o collector do Java fosse também ao mesmo tempo um defragmentador e de fato ele tem uma fase chamada compaction, que move objetos de lugar pra defragmentar se o coletor perceber que o heap acumulou muita fragmentação.

Se não ficou claro, nos diagramas que mostrei até agora, parece que todos os objetos estão movendo de um espaço de memória pra outro. Isso é verdade no Java. Mas num Ruby os objetos não se movem de lugar, os endereços são fixos. Faz de conta que existam listas como uma lista ligada ou um array grande. Cada elemento da lista é um endereço pra memória real, e as listas são as gerações. Então quando eu falo que o objeto é promovido de uma geração pra outra, na memória real nada acontece, só o endereço que sai de uma lista e vai pra outra lista. Num Java, a estratégia de gerações a ajuda a diminuir fragmentação porque à medida que os objetos são promovidos eles movem de lugar na memória e isso vai defragmentando. Mas num Ruby a fragmentação se mantém a mesma, o que vai melhorar é a velocidade do Mark and Sweep.

Esse tipo de processo é o que chamamos de um Compact Copying Collector. A principal função é diminuir a fragmentação da memória. Se você tinha curiosidade, os garbage collectors do Erlang e do .NET são similares. Eles são generational, e tem copy collectors. Os princípios são os mesmos. Só linguagens que lidam com ponteiros e endereços físicos diretamente como as linguagens que compilam nativamente sem um runtime que abstrai a memória ou os interpretadores, não tem como fazer compact ou copy e tendem a fragmentar mais. Não quer dizer que é impossível fazer a parte de copy. Se você mudar um objeto de lugar, ele ganha um novo endereço, agora você precisa garantir que quem tente acessar o endereço antigo caia no novo, algo como uma barreira de leitura. Um dos meus problemas com acesso a ponteiros não é só por aritmética de ponteiro, mas porque dificulta defragmentação via compact copying. Era o que o Go pretendia fazer, mas se não me engano até agora eles não implementaram a fase de copy.

Além disso podemos executar partes dessas fases concorrentemente e paralelamente usando threads reais. Por isso em JVM você ouve falar de nomes como Parallel GC ou ConcurrentMarkSweep GC. O primeiro usa múltiplas threads. O segundo só uma thread, mas ele pausa o mundo só na fase inicial de mark e sweep, mas depois ele roda paralelamente à aplicação. JVM pra servidor usava o Parallel GC por exemplo. Hoje em dia temos o G1GC que usa gerações menores de segmentos 0, 1 e 2 ou Eden, Survivor e Old como expliquei acima, mas ele usa múltiplos segmentos de cada geração em vez de segmentos contíguos e outros algoritmos.

O problema de rodar concorrentemente é a fase de Mark and Sweep, que exige a tal pausa do programa ou Stop the World. Pra evitar isso, linguagens como Erlang implementam uma versão melhor que é mais custosa mas evita a pausa, chamada Tracing collector. Pense uma fase de mark dividida em 3 fases, como se fossem gerações, usando alguma variação do algoritmo chamado Tri-color, onde determinamos cores como preto, cinza e branco e no mark os objetos são movidos pra preto ou branco, e no final quando nenhum objeto for cinza, podemos fazer sweep de todos os objetos que sobraram com a cor branca. A vantagem disso é que podemos fazer essa marcação sem precisar pausar o programa. O Incremental Mark and Sweep do Ruby usa estratégia de Tri-color pro que se chama de Minor GCs ou fases de mark mais rápidas.

O collector do Java tem vários outras estratégias além do que expliquei até agora. De qualquer forma, o que chamamos de um garbage collector, completo, poderia ser um “Parallel Concurrent Generational Incremental Mark and Sweep Copy Compactor Collector”. Ruby e Python fazem algo similar, com a limitação de não poderem ser copy collectors nem rodar collectors concorrentemente. Javascript faz a mesma coisa também.

Generational garbage collectors tendem a usar mais memória pra manter todas as estruturas e segmentos pra cada geração. Ou seja, é um overhead mais alto do que o jeito do Objective-C e Swift. Linguagens compiladas nativamente com um runtime pequeno como C ou Objective-C tendem a depender principalmente do alocador do sistema operacional como o ptmalloc2 ou jemalloc. Linguagens interpretadas são híbridos, ele tem um espaço de heaps alocados com alocador em user-land pra gerenciar os objetos mas usam o alocador do sistema operacional pra blobs maiores como blocos de texto ou arrays grandes.

Uma JVM, se não me engano, pede pro sistema operacional alocar um espaço grande no começo do boot, mas depois ele se vira só em user-land. Vai usar mais memória, mas como não depende de syscalls, tende a ser mais rápido. Ao mesmo tempo, linguagens compiladas e interpretadas dependem do alocador do sistema operacional pra gerenciar a fragmentação de memória como já expliquei no episódio anterior. Uma JVM controla a fragmentação ele mesmo via o processo de copy collector onde ele consegue mover os objetos de lugar.

Falando no alocador do sistema operacional, o ptmalloc2 que é o malloc do glibc que a maioria das linguagens usa, aloca arenas pra cada thread real como já expliquei no episódio anterior. Pior ainda, ele não tem thread-cache ainda como o tcmalloc ou jemalloc. Portanto quanto mais arenas ele alocar, mais memória vai desperdiçar. E o multiplicador pra máquinas 32-bits são de 2 vezes o número de threads reais ou seja, máquinas quad-core com hyperthreading que dá 2 threads por core significa pelo menos 16 arenas. Mas em máquinas 64-bits o multiplicador é 8 então são 8 vezes 4 cores vezes 2 threads que dá 64 arenas!! É muita arena. Em estudos do Heroku que é uma plataforma de serviços para deployment de aplicações em ambiente virtualizado eles recomendam diminuir o teto de arenas. Em máquinas pequenas, com 2 cpus virtuais, em testes empíricos ficou notável que configurar o teto pra 2 arenas é mais que suficiente.

Segundo Hongli Lai, CTO da Phusion, o motivo desse multiplicador ser tão alto pode ser porque o principal desenvolvedor do alocador de memória é da Red Hat que vende produtos e serviços pro mundo enterprise. Lá a prioridade é aumentar performance mesmo que isso custe desperdiçar muita RAM. 10% a mais de performance por 30 vezes mais memória é uma conta que o mundo Enteprise topa pagar. Mas pra gente que precisa balancear performance com uso de memória pra escalar, a recomendação é simples, basta exportar a variável MALLOC\_ARENA\_MAX pra 2 e pronto. Só isso já vai evitar muitos vazamentos de memória por mau uso de arenas.

Gerenciamento de memória é um assunto absurdamente longo. Até este ponto eu expliquei tudo em linhas gerais. Cada parágrafo deste episódio tem uma literatura gigante que você vai precisar procurar pra entender em mais detalhes. Eu falei de copy collectors numa linha, como se fosse algo trivial, mas não é. Eu falei de thread-cache de malloc como se fosse uma correção simples do ptmalloc2 pro tcmalloc, mas não é. Eu falei como o jemalloc é mais rápido que o tcmalloc, mas procure os relatos de Jason Evans pra ver o tanto de trabalho que eles tiveram pra reescrever várias partes, várias vezes, até realmente chegar na versão atual. Alocadores de memória são super fáceis de escrever errado, de um jeito que vai ser super lento e consumir gigabytes de RAM à toa e ficar vazando em servidores. Nunca tente escrever seu próprio alocador a menos que tenha toneladas de dinheiro e tempo pra jogar fora.

Por outro lado, você precisa se interessar em estudar todas as formas que seu alocador funciona. Não basta ver quanto seu processo consome quando sobe. À medida que ele roda o garbage collector vai movendo objetos entre gerações, vai alocando memória do sistema operacional pras suas heaps internas que ele não vai devolver pro sistema operacional ou vai demorar pra devolver. Você precisa achar os pontos no seu programa que geram picos de consumo de memória, porque eles que vão pressionar o tamanho do heap e desperdiçar memória do sistema operacional.

Em última instância, você vai lembrar que todo collector tem dezenas configurações pra fazer tuning. Tamanho inicial do heap, fator de crescimento dos heaps, frequência pra rodar os collectors, quantidade de arenas, etc. Normalmente a configuração que já vem é o suficiente pra 99% das aplicações. Não seja ingênuo de achar que seu programa vai ficar muito melhor fazendo tuning dos alocadores ou garbage collectors, praticamente em todos os casos é seu código que desperdiça memória e tuning nenhum vai resolver isso. Mas no 1% dos casos que você já mediu exaustivamente o comportamento da aplicação em produção com dados reais, já corrigiu todo buraco de código possível, então talvez exista alguma chance de um tuning ajudar um pouco. Foi assim que na comunidade Ruby por exemplo descobrimos sobre o tuning de arenas do ptmalloc2 ou trocar pra jemalloc.

Em ambientes móveis, smartphones, é pior ainda, porque você não tem disco pra fazer swap. Significa que diferente de desktops e servidores, se seus programas foram mal feitos e alocarem memória de pico demais sem desalocar, o OOM ou Out of Memory Killer vai entrar em ação e matar seu programa com muita frequência, tornando a experiência do usuário uma droga. Por isso eu particularmente não gosto das soluções antigas de abrir aplicações que são só casca pra um navegador web e dentro carregar uma aplicação Javascript. No Javascript você não tem controle sobre o garbage collector e não tem acesso ao pool de memória como no num app nativo de iOS onde você pode mexer nos NSAutoreleasePool, significa que você está à mercê de um generational garbage collector, que usa mais memória que um app nativo, vai ter pausas de mark and sweep, e um Just in time compiler que também está alocando memória e usando mais CPU o tempo todo, porque otimizar código não sai de graça. Tudo tem um preço. Quanto mais alguma coisa é conveniente, menos ele tende a ser eficiente.

Quando você está programando pra desktop, que hoje em dia é tudo 64-bits, com muitos endereços virtuais sobrando, mesmo que falte RAM, seu sistema operacional vai jogar tudo pra arquivos de swap em disco. Fica lento, mas nada crasheia, por isso hoje em dia ficou comum fazer todo aplicativo desktop ser uma casca de navegador também e rodar apps web dentro. É o que Spotify, ou Slack ou os editores de texto Atom fazem. Consome memória pra caramba e quanto mais tempo eles ficam rodando mais memória vão consumindo. Por outro lado são super convenientes de fazer e você não precisa controlar memória manualmente. Eu ainda tenho preferência por programas nativos bem feitos que alocam memória de formas eficientes sem a necessidade de usar um garbage collector custoso. Um garbage collector é conveniente, mas não é necessário, especialmente em programas desktop.

Como eu já disse antes, tudo é uma questão de trade-off, você troca conveniência por mais lentidão e mais memória. Quanto menos conveniente for, mais você ganha performance e usa menos memória. Não existe um vencedor óbvio, cada caso é um caso. É um aplicativo pra servidor? Pra desktop? Pra smartphone? É grande? É pequeno? Precisa trabalhar quantidades grandes de dados com muita frequência? Precisa coordenar o processamento desses dados paralelamente? Ou de forma distribuída? Tudo isso afeta o que você vai escolher.

Com isso você também deve entender porque eu disse no episódio passado que linguagens com máquinas virtuais como Java ou Erlang preferem tomar conta de todos os recursos da máquina, porque o ideal é deixar eles alocarem toda a memória disponível pra gerenciarem os recursos internamente com seus garbage collectors. Percebam que eles levam pra user-land todas as atribuições do sistema operacional no kernel space. Eles fazem poucas syscalls porque conseguem gerenciar coisas como corotinas e gerenciar memória tudo em user-land.

O Go usa as funções de alocação como um malloc faz e prefere ficar mais baixo nível, mas ao mesmo tempo evitando depender demais do sistema operacional. Ele evita um pouco da fragmentação porque divide a memória em slabs dependendo do tamanho dos chunks como o ptmalloc e outros mallocs fazem. Mas diferente de Java e Erlang ele não tem e vai ter trabalho pra implementar um copy collector porque expõe ponteiros. É o mesmo motivo de porque Python ou Ruby também Não tem copy collectors. Generalizando um pouco, linguagens interpretadas ou com máquinas virtuais tendem a usar mais memória. Linguagens que não expõe ponteiros podem implementar compact copying collectors e evitar mais fragmentação ao custo de usar mais processamento.

E finalmente, chegamos ao último episódio do tema de Back-end! Gerenciamento de Memória era o último assunto pra vocês finalmente terem mais entendimento do funcionamento da máquina e sistema operacional. A discussão sobre linguagens só faz sentido se você entende essas coisas, senão a discussão fica mais parecendo fofoca de telenovela, sem argumentos e só repetindo frases de efeito que alguém disse num tweet ou blog post. Todo mundo só sabe até o ponto que a linguagem tem ou não garbage collector. E todo mundo só acredita que ter um generational garbage collector é bom. Mas ninguém sabe explicar porque é bom e o que você paga tem troca.

Mais do que isso, sem entender concorrência, paralelismo e gerenciamento de memória, você também não consegue entrar direito em devops e lá também vai ficar só seguindo superstições, mandingas, e repetindo o que ouviu os outros falar sem realmente saber o que está fazendo. Entender como a máquina e o sistema operacional realmente funcionam, pra mim, é o que começa a dividir as crianças dos adultos.

Se você tem dúvida mande nos comentários abaixo, se curtiu mande um joinha, compartilhe com seus amigos, não esqueça de assinar o canal e clicar no sininho pra não perder os próximos episódios. A gente se vê semana que vem, até mais.